



## 2. Platforma Microsoft .NET

Maciej Piechówka  
macpi@eti.pg.gda.pl

### Spis treści:

#### 2.1 Microsoft .NET Framework: CLR, podzespoły, biblioteka klas, elementy C#

#### 2.3 Dostęp do danych

- ADO.NET: tryb dostępu, aktualizacja, procedury składowane, transakcje, integracja z XML
- LINQ, ADO.NET Entity Framework, ADO.NET Data Services

#### 2.4 Wytwarzanie aplikacji .NET: Podstawy ASP.NET

- Model strony, WebForms,
- Model delegacyjny zdarzeń,
- Kontrolki serwerowe, użytkownika, sprawdzające, wiązanie danych,
- ASP.NET od wewnątrz
- Zarządzanie stanem

#### 2.5 Tworzenie i korzystanie z usług Web (*web services*), użycie protokołów SOAP i UDDI

##### Literatura:

- Materiały firmy Microsoft, Piotr Bubacz *ITA-103 Aplikacje Internetowe*, zasoby Internetu...
- Platt D. *Podstawy Microsoft .Net*, RM, 2001
- Pery C. s. *Core C# i .NET*, Helion 2006
- Esposito D. *Tworzenia aplikacji za pomocą ASP.NET oraz ADO.NET*, RM 2002
- Connolly R. *ASP.NET 2.0. Projektowanie aplikacji internetowych*, Helion 2008
- Matulewski J., *C# 3.0 i .NET 3.5. Technologia LINQ*, Helion, 2008



## 2.3.5 LINQ - Language Integrated Query

- unifies data access
- simplified querying objects, data and XML by integrating query and transform operations into the programming language
- lets you query any collection implementing *IEnumerable<>*, whether an array, list, XML DOM, or remote data source (such as a table in SQL Server).
- introduces a concise declarative syntax (similar to SQL) that is consistent irrespective of what your underlying data source is: in memory custom objects, datasets, XML, Entities etc
- offers the benefits of both compile-time type checking and dynamic query composition.

### LINQ DLLs and namespaces:

*System.Core.dll*

*System.Linq* - standard query operators, *System.Linq.Expressions*

*System.Data.Linq.dll*

*System.Data.Linq* - LINQ to SQL, *System.Data.Linq.Mapping*, *System.Data.Linq.SqlClient*

*System.Data.DataSetExtensions.dll*

*System.Data* - LINQ to DataSet

*System.Xml.Linq.dll*

*System.Xml.Linq* - LINQ to XML

[LINQ: .NET Language-Integrated Query](http://msdn.microsoft.com/en-us/library/bb308959.aspx)  
<http://msdn.microsoft.com/en-us/library/bb308959.aspx>



## 2.3.5.1 LINQ Architecture, Query expressions

Visual C#

Visual Basic

Others

.Net Language Integrated Query (LINQ)

LINQ-enabled data sources

LINQ-enabled ADO.NET

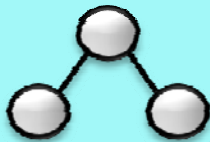
LINQ  
To Objects

LINQ  
To Datasets

LINQ  
To SQL

LINQ  
To Entities

LINQ  
To XML



Objects



Databases



XML



## Relational Data with ADO.Net

```
List<Customer> customers = new List<Customer>();
SqlConnection c = new SqlConnection(ConnectionString);
SqlCommand cmd = new SqlCommand(
    @"SELECT c.Name, c.Phone, c.ID
    FROMM Customers c
    WHERE c.City = @po");
cmd.Parameters.AddWithValue("@p0", "USA");
DbDataReader dr = cmd.ExecuteReader();
while (dr.Read())
{
    Customer cust = new Customer();
    cust.CompanyName = dr.GetString(0);
    cust.ContactName = dr.GetString(1);
    cust.Country = dr.GetString(2);
    cust.CustomerID = dr.GetString(3);
}
dr.Close();
return customers;
```

### Errors:

- Syntax error in SQL – “FROMM”
- Mismatch on parameter name – “@po” vs. “@p0” (o vs. zero)
- command not associated with connection
- connection not opened
- connection not disposed
- Items not added to the customer list in the iteration
- Fields fetched don't agree with fields in the select clause (3 fields in select, 4 properties read)
- Possible data type mismatch between the ID and CustomerID



# Manipulating data with Objects

```
List<Customer> custs = Customer.GetCustomers();  
//Sort by company name, filter by country, total  
orders  
SortedList<string, CustOrders> ordered = new  
    SortedList<string, CustOrders>();  
foreach (Customer cust in custs)  
    if (cust.Country.Contains(Filter.Text))  
    {  
        CustOrders item = new CustOrders();  
        decimal orderTotal = 0;  
        foreach (Order order in cust.Orders)  
            orderTotal += order.Amount;  
        item.CompanyName=cust.CompanyName;  
        item.ContactName=cust.ContactName;  
        item.TotalAmount=orderTotal;  
        ordered.Add(cust.CompanyName, item);  
    }
```



# Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;
        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.WriteLine("{0,1} ", num);
        }
    }
}
```



## LINQ - przykład

```
static void Main(string[] args)
{
    var results =
        from p in Process.GetProcesses()
        select p;
    //
    foreach (var o in results)
    {
        Console.WriteLine(o.ToString());
    }
    //
    Console.ReadLine();
}
```

from p in Process.GetProcesses()  
where p.Threads.Count > 6  
select p;

from p in Process.GetProcesses()  
where p.Threads.Count > 6  
orderby p.ProcessName descending  
select p;

from p in Process.GetProcesses()  
where p.Threads.Count > 6  
orderby p.ProcessName descending  
select new {p.ProcessName, ThreadCount = p.ThreadsCount}

```
...
FileInfo[] files;
IEnumerable sorted = null;
...
sorted = from file in files
         orderby file.Name ascending
         select file;
```



# LINQ Style programming

```
class Contact { ... };  
List<Contact> contacts = new List<Contacts>();  
foreach(Customer c in customers)  
{  
    if(c.State == "WA")  
    {  
        Contact ct = new Contact();  
        ct.Name = c.Name;  
        ct.Phone = c.Phone;  
        contacts.Add(ct);  
    }  
}
```

```
var contacts =  
    from c in customers  
    where c.State == „Gdańsk”  
    select new { c.Name, c.Phone };
```



## Zastosowanie rozszerzeń C# 3.0

```
var contacts =  
    from c in customers  
    where c.State == „Gdansk”  
    select new { c.Name, c.Phone };
```

Query  
expressions

Local variable  
type inference

```
var contacts =  
    customers  
    .Where(c => c.State == „Gdansk”)  
    .Select(c => new { c.Name, c.Phone });
```

Lambda  
expressions

Extension  
methods

Anonymous  
types

Object  
initializers



## Data source

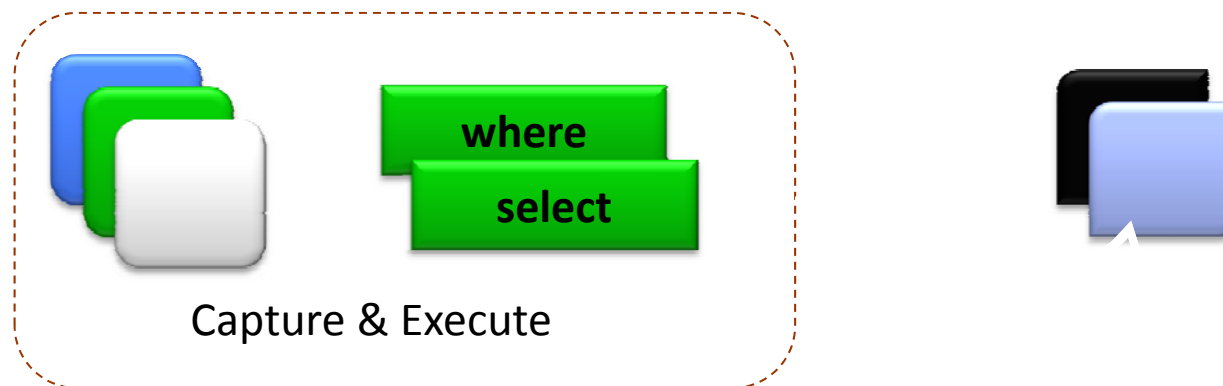
### Queryable types: `IEnumerable<T>` & `IQueryable<T>`

Types that support `IEnumerable<Of <(T)>>` or a derived interface such as the generic `IQueryable<Of <(T)>>` serve as a **LINQ data source**.

- **`IEnumerable<Of <(T)>>`** is the interface that enables generic collection classes to be enumerated by using the *foreach* statement.



- **`IQueryable`** – query executed in one go
  - build an expression tree that represents the query to be performed.





# Solution: Extension Methods

```
namespace System.Query
{
    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
            Func<T, bool> predicate) { ... }

        public static IEnumerable<S> Select<T, S>(this IEnumerable<T> source,
            Func<T, S> selector) { ... }

        ...
    }
}
```

Extension  
methods

```
using System.Query;
```

```
IEnumerable<string> contacts =  
    customers.Where(c => c.State == "WA").Select(c => c.Name);
```

IntelliSense!



# Query expressions

**A query expression is** a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid.

**A query expression** consists of a **set of clauses** written in a declarative syntax similar to SQL or XQuery. Each clause in turn contains one or more C# expressions, and these expressions may themselves be either a query expression or contain a query expression.

**A query expression** must begin with a **from** clause and must end with a **select** or **group** clause.

Between the first **from** clause and the last **select** or **group** clause, it can contain one or more of these optional clauses: **where**, **orderby**, **join** and additional **from** clauses. You can also use the **into** keyword to enable the result of a **join** or **group** clause to serve as the source for additional query clauses in the same query expression.

<http://msdn.microsoft.com/en-us/library/bb308959.aspx>

LINQPad: [www.linqpad.net](http://www.linqpad.net)



# Query Expressions

- Introduce SQL-Like Syntax to Language
- Compiled to Traditional C# (via Extension Methods)

Starts with  
**from**

Zero or more **from**,  
**join**, **let**, **where**, or  
**orderby**

```
from id in source  
{ from id in source |  
  join id in source on expr equals expr [ into id ] |  
  let id = expr |  
  where condition |  
  orderby ordering, ordering, ... }  
select expr | group expr by key  
[ into id query ]
```

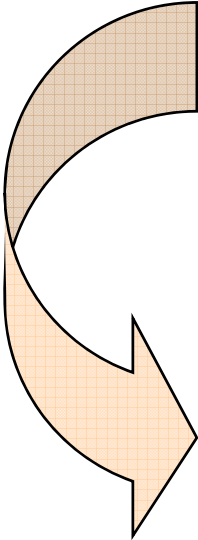
Ends with **select**  
or **group by**

Optional **into**  
continuation



# Query Expressions

- Queries translate to method invocations
  - Where, Join, OrderBy, Select, GroupBy, ...



```
from c in customers
where c.State == „Gdańsk”
select new { c.Name, c.Phone };
```

```
customers
.Where(c => c.State == „Gdańsk”)
.Select(c => new { c.Name, c.Phone });
```



# Standard Query Operators

The methods that form the LINQ pattern. They can be used to query any data source for which a LINQ provider is available.

The *Standard Query Operators* is enables querying of any .NET array or collection. The Standard Query Operators consists of the extension methods.

The Standard Query Operators operate on *sequences*. Any object that implements the interface *IEnumerable<T>* for some type T is considered a sequence of that type.

*f*



# Standard Query Operators ...

## Types of Standard Query Operators:

- **Restriction Operators - *Where***
- **Projection Operators - *Select, SelectMany***
- **Partitioning Operators - *Skip, SkipWhile, Take, TakeWhile***
- **Join Operators - *Join, GroupJoin***
- **Concatenation Operators - *Concat***
- **Ordering Operators - *OrderBy, OrderByDescending, Reverse, etc.***
- **Grouping Operators - *GroupBy***
- **Set Operators - *Distinct, Except, Intersect, Union***
- **Conversion Operators - *Cast, OfType, ToArray, ToList, ToDictionary, etc.***
- **Equality Operators - *SequenceEqual***
- **Element Operators - *DefaultEmpty, First, Last, Single, ElementAt, etc.***
- **Generation Operators - *Empty, Range, Repeat***
- **Aggregate Operators - *Average, Count, Max, Min, Sum, etc.***
- **Quantifiers – *Any, All, Contains***



# LINQ Standard Query Operators

## Restriction Operators

Where	Enumerates the source sequence and yields those elements for which the predicate function returns true
-------	--

## Projection Operators

Select	Enumerates the source sequence and yields the results of evaluating the selector function for each element
SelectMany	Performs a one-to-many element projection over a sequence

## Partitioning Operators

Skip	Skips a given number of elements from a sequence and then yields the remainder of the sequence.
SkipWhile	Skips elements from a sequence while a test is true and then yields the remainder of the sequence. Once the predicate function returns false for an element, that element and the remaining elements are yielded with no further invocations of the predicate function.
Take	Yields a given number of elements from a sequence and then skips the remainder of the sequence
TakeWhile	Yields elements from a sequence while a test is true and then skips the remainder of the sequence. Stops when the predicate function returns false or the end of the source sequence is reached

```
IEnumerable<Order> orders = customers
    .Where(c => c.Country == "Denmark")
    .SelectMany(c => c.Orders);
```

```
IEnumerable<Product> MostExpensive10 =
    products.OrderByDescending(p => p.UnitPrice).Take(10);
```



# LINQ Standard Query Operators (2)

## Join Operators

Join	Performs an inner join of two sequences based on matching keys extracted from the elements
GroupJoin	Performs a grouped join of two sequences based on matching keys extracted from the elements

```
var custOrders = customers
    .Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c.Name, o.OrderDate, o.Total } );
```

## Concatenation Operators

Concat	Enumerates the first sequence, yielding each element, and then it enumerates the second sequence, yielding each element.
--------	--

## Ordering Operators

OrderBy, OrderByDescending, ThenBy, ThenByDescending	Make up a family of operators that can be composed to order a sequence by multiple keys.
Reverse	Reverses the elements of a sequence

```
IEnumerable<Product> orderedProducts = products
    .OrderBy(p => p.Category)
    .ThenByDescending(p => p.UnitPrice)
    .ThenBy(p => p.Name);
```



# LINQ Standard Query Operators (3)

## Grouping Operators

GroupBy	Groups the elements of a sequence
---------	-----------------------------------

```
IEnumerable<string> productCategories =  
products.Select(p => p.Category).Distinct();
```

## Set Operators

Distinct	Eliminates duplicate elements from a sequence.
Except	Enumerates the first sequence, collecting all distinct elements; then enumerates the second sequence, removing elements contained in the first sequence.
Intersect	Enumerates the first sequence, collecting all distinct elements; then enumerates the second sequence, yielding elements that occur in both sequences.
Union	Produces the set union of two sequences.

## Conversion Operators

AsEnumerable	Returns its argument typed as <code>IEnumerable&lt;T&gt;</code>
Cast	Casts the elements of a sequence to a given type.
OfType	Filters the elements of a sequence based on a type
ToArray	Creates an array from a sequence.
ToDictionary	Creates a <code>Dictionary&lt;TKey, TElement&gt;</code> from a sequence (one-to-one).
ToList	Creates a <code>List&lt;T&gt;</code> from a sequence
ToLookup	Creates a <code>Lookup&lt;TKey, TElement&gt;</code> from a sequence (one-to-many)



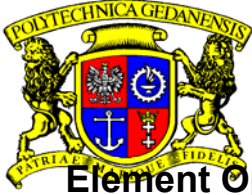
# LINQ Standard Query Operators (4)

```
string[] customerCountries = customers
    .Select(c => c.Country).Distinct().ToArray();
List<Customer> customersWithOrdersIn2005 = customers
    .Where(c => c.Orders.Any(o => o.OrderDate.Year == 2005)).ToList();
Dictionary<string,decimal> categoryMaxPrice = products
    .GroupBy(p => p.Category)
    .ToDictionary(g => g.Key, g => g.Max(p => p.UnitPrice));
ILookup<string,Product> productsByCategory = products
    .ToLookup(p => p.Category);
IEnumerable<Product> beverages = productsByCategory["Beverage"];
List<Person> persons = GetListOfPersons();
IEnumerable<Employee> employees = persons.OfType<Employee>();
ArrayList objects = GetOrders();
IEnumerable<Order> ordersIn2005 = objects
    .Cast<Order>()
    .Where(o => o.OrderDate.Year == 2005);
```

## Equality Operators

SequenceEqual

Checks whether two sequences are equal by enumerating the two source sequences in parallel and comparing corresponding elements



# LINQ Standard Query Operators (5)

## Element Operators

DefaultIfEmpty	Supplies a default element for an empty sequence. Can be combined with a grouping join to produce a left outer join
ElementAt	Returns the element at a given index in a sequence
ElementAtOrDefault	Returns the element at a given index in a sequence, or a default value if the index is out of range
First	Returns the first element of a sequence
FirstOrDefault	Returns the first element of a sequence, or a default value if no element is found
Last	Returns the last element of a sequence
LastOrDefault	Returns the last element of a sequence, or a default value if no element is found
Single	Returns the single element of a sequence. An exception is thrown if the source sequence contains no match or more than one match.
SingleOrDefault	Returns the single element of a sequence, or a default value if no element is found

```
Customer customer = customers.First(c => c.Phone == "111-222-3333");
```

```
Customer customer = customers.Single(c => c.CustomerID == 1234);
```

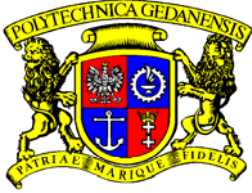
## Generation Operator

Empty	Returns an empty sequence of a given type
Range	Generates a sequence of integral numbers
Repeat	Generates a sequence by repeating a value a given number of times

```
int[] squares = Enumerable.Range(0, 100).Select(x => x * x).ToArray();
```

```
long[] allOnes = Enumerable.Repeat(-1L, 256).ToArray();
```

```
IEnumerable<Customer> noCustomers = Enumerable.Empty<Customer>();
```



# LINQ Standard Query Operators (6)

## Quantifiers

Any	Checks whether any element of a sequence satisfies a condition. If no predicate function is specified, simply returns true if the source sequence contains any elements. Enumeration of the source sequence is terminated as soon as the result is known
All	Checks whether all elements of a sequence satisfy a condition. Returns true for an empty sequence.
Contains	Checks whether a sequence contains a given element

```
bool b = products.Any(p => p.UnitPrice >= 100 && p.UnitsInStock == 0);  
IEnumerable<string> fullyStockedCategories = products  
    .GroupBy(p => p.Category)  
    .Where(g => g.All(p => p.UnitsInStock > 0))  
    .Select(g => g.Key);
```

## Aggregate Operators

Aggregate	Applies a function over a sequence
Average	Computes the average of a sequence of numeric values
Count, LongCount	Counts the number of elements in a sequence
Max	Finds the maximum of a sequence of numeric values
Min	Finds the minimum of a sequence of numeric values.
Sum	Computes the sum of a sequence of numeric values.



## 2.3.5.1 LINQ to Objects

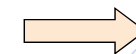
```
using System;
using System.Linq;
using System.Collections.Generic;

class app {
    static void Main() {
        string[] names = { "Burke", "Connor",
                           "Frank", "Everett",
                           "Albert", "George",
                           "Harris", "David" };

        Func<string, bool> filter = s => s.Length == 5;
        Func<string, string> extract = s => s;
        Func<string, string> project = s => s.ToUpper();

        IEnumerable<string> expr = names
            .Where(filter).OrderBy(extract)
            .Select(project);

        foreach (string item in expr)
            Console.WriteLine(item);
    }
}
```



BURKE  
DAVID  
FRANK

- Query Operators can be used against any .NET collection (***IEnumerable<T>***)
  - Select, Where, GroupBy, Join, etc.
- Deferred Query Evaluation
- Lambda Expressions



# LINQ To In-Memory Objects

LINQ To In-Memory Objects is used to write queries against in-memory collections and other queryable sources in any .NET language using LINQ.

**Example:**

Let us consider a class **Person** as below:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

Now create and populate a **collection of persons** using **Person** class

```
//Create a list of person
System.Collections.Generic.List<Person> IstPerson =
    new System.Collections.Generic.List<Person>
{
    new Person { FirstName = "A", LastName = "X", Age = 15 },
    new Person { FirstName = "B", LastName = "Y", Age = 17 },
    new Person { FirstName = "C", LastName = "Z", Age = 24 }
};
```



## LINQ To In-Memory Objects - przykład

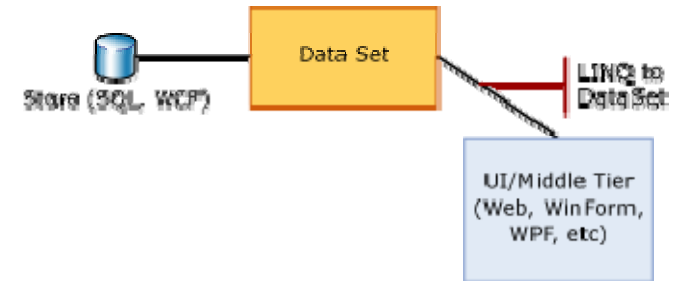
We could use the standard "**Where()**" extension method provided by *System.Linq* to retrieve a sequence of those "Person" objects within this collection whose **FirstName** starts with the letter "A" as:

```
IEnumerable<Person> queryResult;  
queryResult = IstPerson.Where(p => p.FirstName.StartsWith("A"));  
  
Response.Write(queryResult.Count().ToString()); //Total result count  
Response.Write(queryResult.First().FirstName); //Display First Name  
  
Response.Write(queryResult.First().LastName); //Display Last Name  
  
//Compute average age of person in list and display.  
Response.Write(IstPerson.Average(p => p.Age).ToString());  
  
//Compute max age of person in list and display.  
Response.Write(IstPerson.Max(p => p.Age).ToString());
```



## 2.3.5.2 LINQ to DataSet

- ***DataSets*** fully integrated with LINQ
- Both typed and un-typed ***DataSets*** will work
- Do joins, grouping, etc over Data in ***DataTables***
- Can create view that span multiple ***DataTables***



### Example:

#### // Query Expression Syntax

```
DataSet ds = new DataSet();  
FillTheDataSet(ds); //Fill the DataSet.  
DataTable dtPeople = ds.Tables["People"];  
IEnumerable<DataRow> query =  
    from people In dtPeople.AsEnumerable()  
    select people;  
foreach (DataRow p in query)  
    Response.Write(p.Field<string>("FirstName"));
```

The `AsEnumerable` method must be called to get the `IEnumerable<DataRow>` interface

#### [LINQ to DataSet](http://msdn.microsoft.com/en-us/library/bb386977.aspx)

<http://msdn.microsoft.com/en-us/library/bb386977.aspx>



# LINQ To DataSet - Method-Based Query Syntax

## Method-Based Query Syntax

The method-based query syntax is a sequence of direct method calls to LINQ operator methods, passing lambda expressions as the parameters.

### Example:

```
DataSet ds = new DataSet();
FillTheDataSet(ds); //Fill the DataSet.
DataTable dtPeople = ds.Tables["People"];
var query = dtPeople.AsEnumerable().
    Select(people => new {
        FirstName = people.Field<string>("FirstName"),
        LastName = people.Field<string>(" LastName"),
        Age = people.Field<int>("Age")
    });
foreach (var personInfo in query)
{
    Response.Write(personInfo.FirstName );
}
```

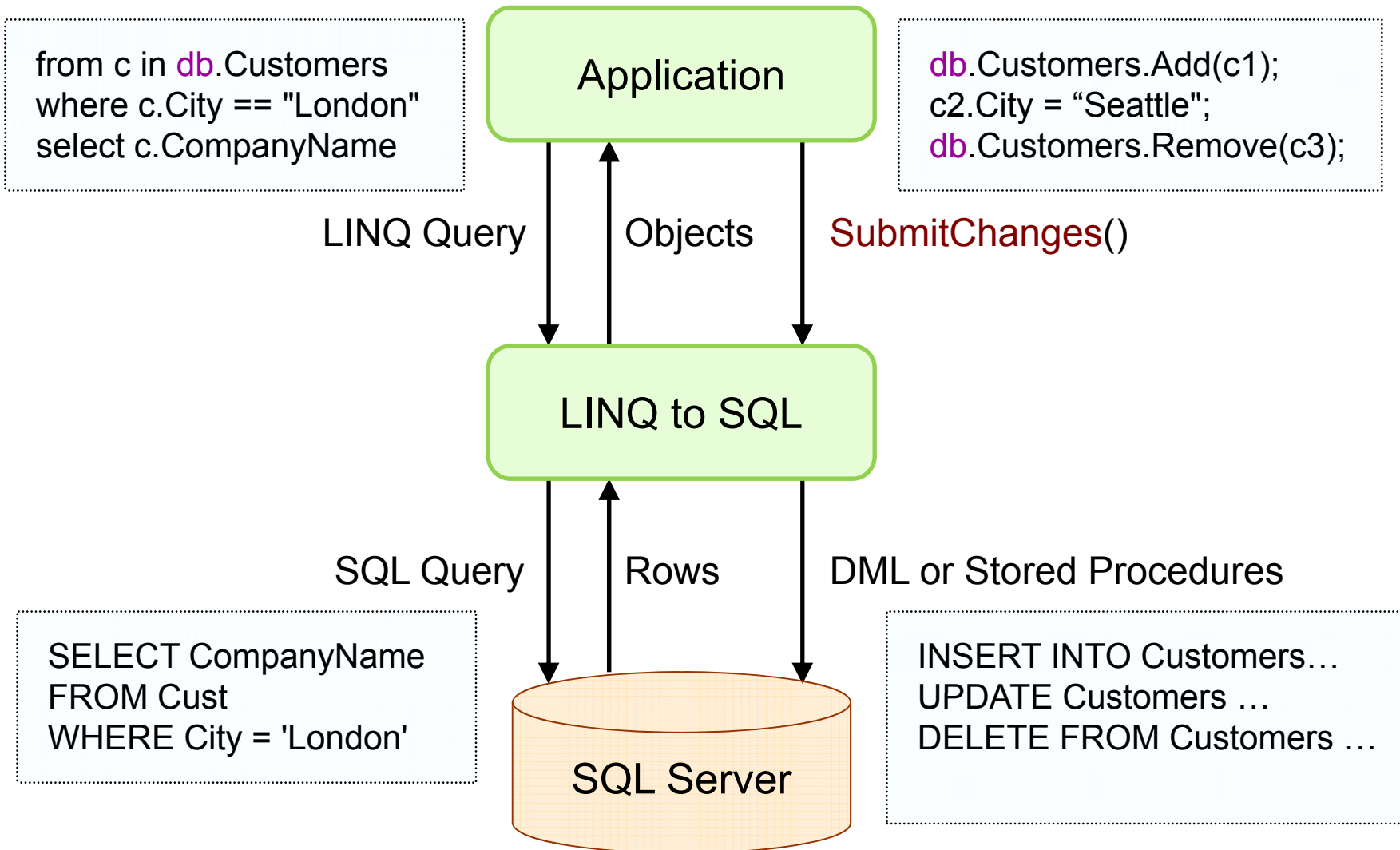


## 2.3.5.3 LINQ to SQL

- Provides object-to-relational mapping (ORM) within the .NET Framework for Microsoft SQL Server databases
  - Access relational data as strongly typed objects using LINQ query expressions
  - Language integrated data access
    - Maps tables and rows to classes and objects
    - Builds on ADO.NET and .NET Transactions
  - Mapping
    - Encoded in attributes or externally
    - Tool generated or manually authored
    - Relationships map to **properties**
  - Persistence
    - Automatic change tracking
    - Updates through SQL or stored procedures
- ORM
  - Entity Classes
  - Creating Entity Classes
  - DataContext Class
  - Submitting Queries to Relational Databases
  - Invoking Stored Procedures

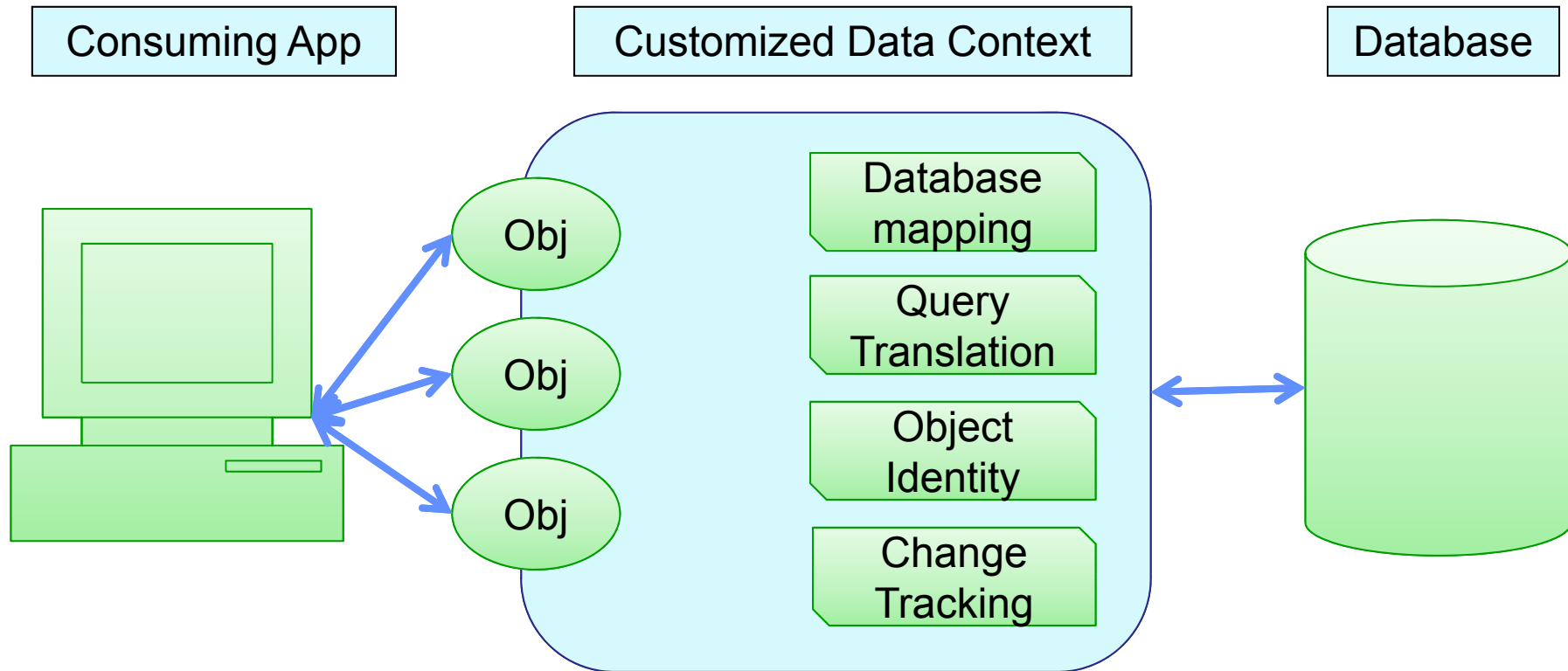


# LINQ to SQL





# DataContext services





## LINQ approach

*// First we will declare instance of **DataContext** class.*  
*// It is responsible for the **translation of LINQ to SQL**, and*  
*// the **mapping** of the results (**rows**) of that query **to objects**.*  
*// The **DataContext** can be equated to a database, in that it contains a series of tables*  
*// and stored procedures and views.*

```
CustomersDataContext dbCust = new CustomersDataContext();
```

```
var query = from p in dbCust.Customers
```

```
    where p.City == "Paris"
```

```
    select p.Name, p.Country;
```

```
foreach (var cust in query)
```

```
{
```

```
    Response.Write(cust.Name);
```

```
    Response.Write(cust.Country);
```

```
}
```



# LINQ To SQL - Role of Entity Classes

LINQ to SQL enable us to model classes that **map** to/from a database. These classes are referred to as "**Entity Classes**" and instances of them are called "Entities".

- **Entity classes map to tables** within a database.
- The properties of entity classes map to the table's columns.
- Each **instance of an entity class** then represents a **row** within the database table.

All classes created using the LINQ to SQL designer are defined as "**partial classes**" - which means we can optionally drop into code and add additional properties, methods and events to them.



# LINQ to SQL - Defining Entity Classes

- Accessing data with LINQ to SQL

```
[Table(Name="Customers")  
public class Customer {  
    [Column(DbType="nvarchar(32) not null", Id=true)]  
    public string ContactName{...}  
    ...  
}
```

Tables are like collections

Classes describe data

```
DataContext context = new DataContext(  
    "Initial Catalog=petdb;Integrated Security=sspi");
```

Strongly typed connection

```
Table<Customer> custs = context.GetTable<Customer>();
```

Define the potential to interact with DB

```
var query = from c in custs  
            where c.City = "London"  
            select new { c.Name, c.Phone };
```

Integrated query syntax



# LINQ To SQL - Object Relational Designer Tool

## Creating Entity Classes

### Object Relational Designer Tool

This designer provides a rich user interface for creating an object model from an existing database. This tool is part of the Visual Studio IDE.

**LINQ to SQL Class** – this is a **file template** that lives in Visual Studio 2008. We can create and edit LINQ to SQL classes by using the Visual Studio 2008 designer tool.

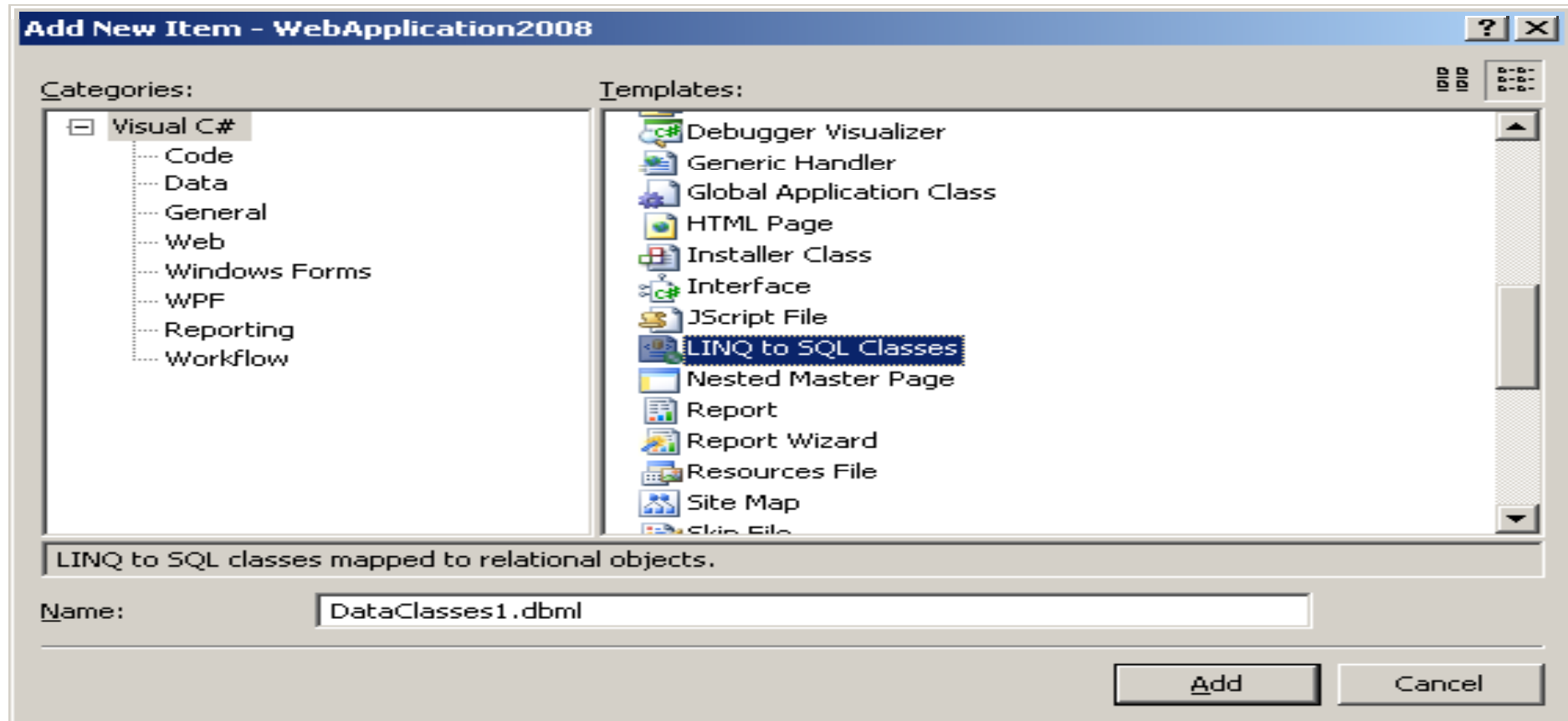
To add a LINQ to SQL Class file to a project:

- From within a C# or VB application, on the Project menu, click Add New Item.
- Click the **LINQ to SQL Classes** template.
- Either provide an alternative name or keep the default name of *DataClasses1.dbml*.
- Click Add.



# LINQ To SQL template in VS

Visual Studio 2008 designer to create LINQ to SQL Class





## LINQ to SQL Class – *DataContext* creation

### *LINQ to SQL Class*

The **.dbml** file is added to the project and the O/R Designer opens. The name provided will be the name of the generated *DataContext*.

E.g., using the default name will cause the designer to name the **DataClasses1DataContext**.

After you add a LINQ to SQL file to your project, the empty design surface represents a *DataContext* ready to be configured.

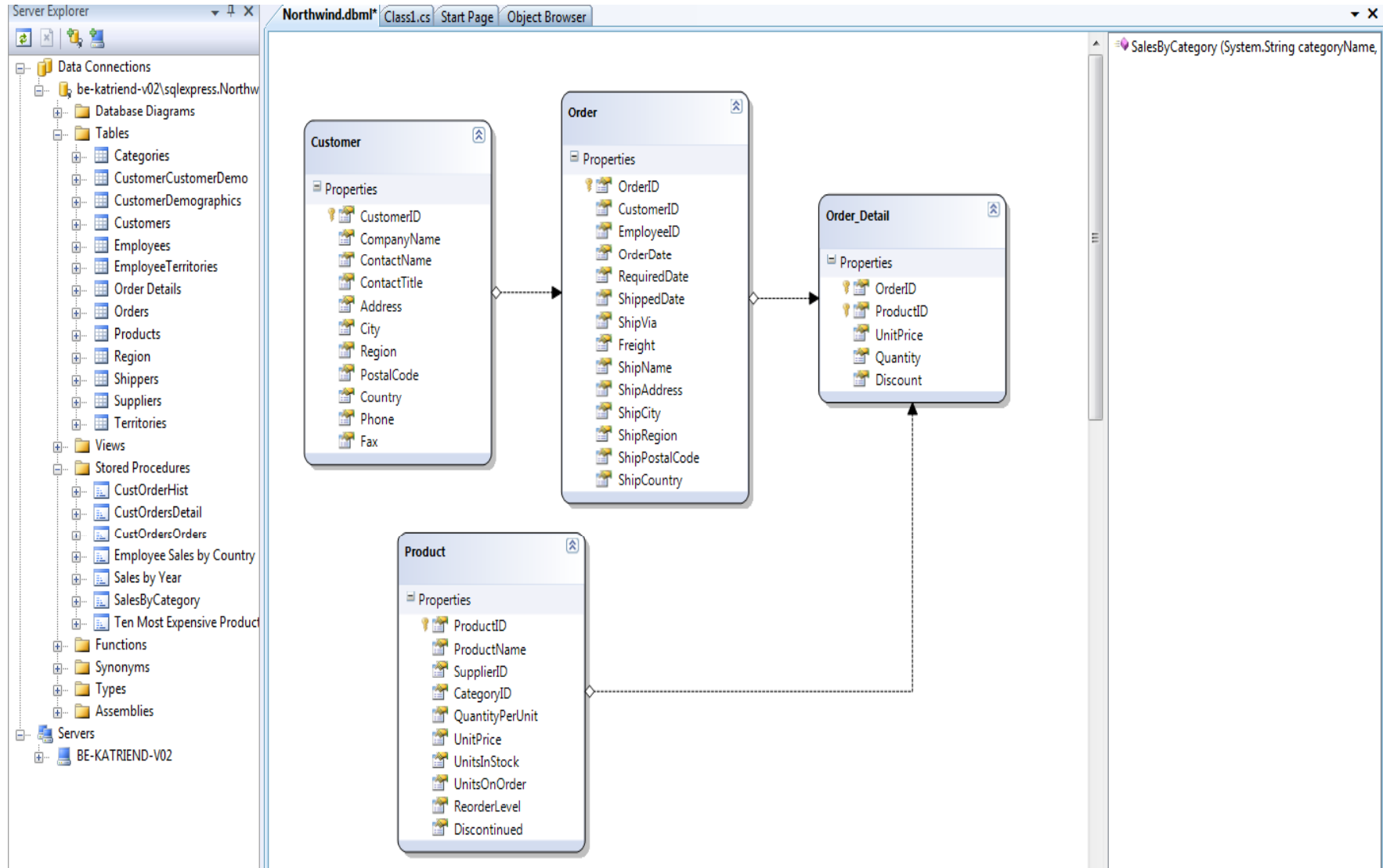
**Drag database** items from Server Explorer/Database Explorer onto the O/R Designer to create data classes and DataContext methods.

The data connection of a *DataContext* is created based on the first item added to the designer from Server Explorer/Database Explorer.

The created *DataContext* class is derived from *System.Data.Linq.DataContext*.



# LINQ to SQL Overview





# LINQ To SQL - DataContext Class

***DataContext Class*** - represents the main entry point for the LINQ to SQL framework.

The **DataContext** is

- the source of all entities **mapped** over a database connection.
- **tracks changes** that you made to all retrieved entities and
- **maintains an "identity cache"** that guarantees that entities retrieved more than one time are represented by using the **same object instance**.

The "**LINQ to SQL Class**" file allow us to model classes that represent a relational database.

It will also create a strongly-typed "**DataContext**" class that will have **properties** that represent each *Table* we modeled within the database, as well as **methods** for each *Stored Procedure* we modeled.

**DataContext** class is the main conduit by which **we'll query entities** from the database as well as apply changes back to it.



# LINQ To SQL - select query

**Select Query:** selecting rows from table is achieved by just writing a LINQ query in our own programming language, and then executing that query to retrieve the results. LINQ to SQL itself translates all the necessary operations into the necessary SQL operations.

## Example:

```
PersonDataClassesDataContext dbPeople = new PersonDataClassesDataContext();  
var query = from p in dbPeople.Peoples  
            where p.Age > 18  
            select p;  
foreach (var ppl in query)  
{  
    Response.Write(ppl.FirstName);  
}
```



# LINQ To SQL - insert

**Insert Query:** to execute a SQL Insert, just add objects to the object model we have created, and call ***SubmitChanges*** on the created ***DataContext***.

```
PersonDataClassesDataContext dbPeople = new PersonDataClassesDataContext();  
    People objPeople = new People();  
    objPeople.FirstName = "Gyan";  
    objPeople.LastName = "Singh";  
    objPeople.Age = 33;  
    dbPeople.Peoples.InsertOnSubmit(objPeople);  
    // At this point, the new People object is added in the object model.  
    // In LINQ to SQL, the change is not sent to the database until SubmitChanges is called.  
    dbPeople.SubmitChanges();
```



## LINQ To SQL - update

**Update Query:** to Update a database entry, first retrieve the item and edit it directly in the object model. After you have modified the object, call ***SubmitChanges*** on the ***DataContext*** to update the database.

### Example:

```
PersonDataClassesDataContext dbPeople = new PersonDataClassesDataContext();  
    var query = from p in dbPeople.Peoples  
                select p;  
    var intAge = 18;  
    foreach (var ppl in query) {  
        ppl.Age = intAge;  
        intAge++;  
    }  
    dbPeople.SubmitChanges();
```



## LINQ To SQL - delete

**Delete Query:** to Delete an item, remove the item from the collection to which it belongs, and then call *SubmitChanges* on the *DataContext* to commit the change.

### Example:

the person whose PersonID is 1 retrieved from the database. Then, after confirming that the people row was retrieved, *DeleteOnSubmit* is called to remove that object from the collection. Finally, *SubmitChanges* is called to forward the deletion to the database.

```
PersonDataClassesDataContext dbPeople = new PersonDataClassesDataContext();  
var query = from p in dbPeople.Peoples  
            where p.PersonID == 1  
            select p;  
if (query.Count() > 0)    {  
    dbPeople.Peoples.DeleteOnSubmit(query.First());  
    dbPeople.SubmitChanges();  
}
```



## 2.3.5.4 LINQ to XML - *System.Xml.Linq*

Language integrated query for XML:

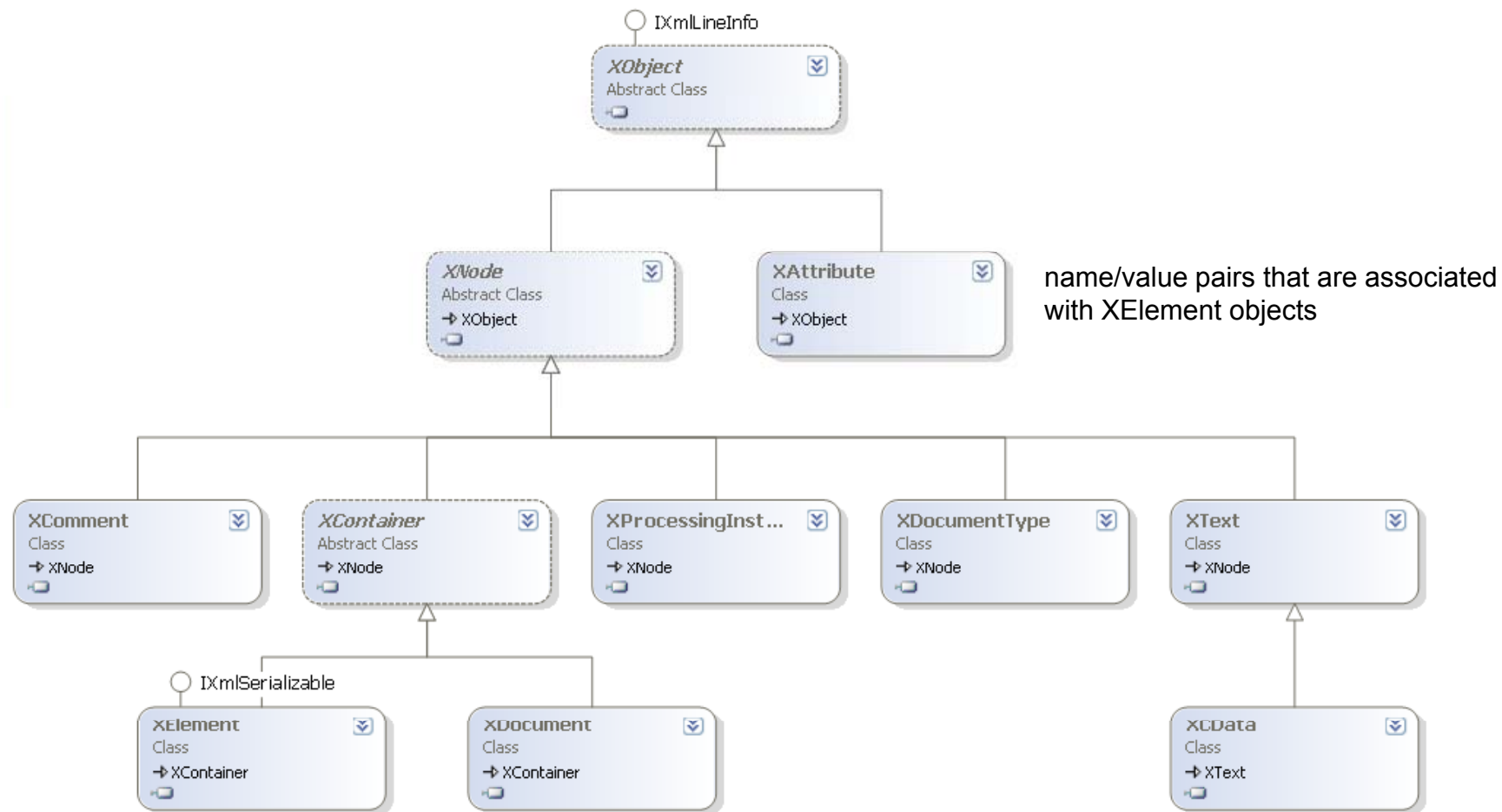
- designed to make XML usable without learning specialized technologies such as XPath/XSLT
- builds on top the standard LINQ query commands
- enables you to write queries on the in-memory XML document to retrieve collections of elements and attributes.
- compose query expressions with multiple data stores
- give ability to use query results as parameters to *XElement* and *XAttribute* object constructors enables a powerful approach to creating XML trees

### LINQ to XML

- provides much richer (and easier) querying and data shaping support than the low-level *XmlReader/XmlWriter* API in .NET today.
- more efficient (and uses much less memory) than the DOM API that *XmlDocument* provides.



# LINQ to XML Class Hierarchy



XML element nodes that contain other elements as child nodes.  
Methods: *Load*, *Parse*, *Save* and *WriteTo*,  
...

complete XML document

<http://msdn.microsoft.com/en-us/library/bb308960.aspx>  
<http://msdn.microsoft.com/en-us/library/bb387087.aspx>



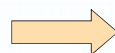
## LINQ to XML - example

```
TodayXmlDocument doc = new XmlDocument();
XmlElement contacts = doc.CreateElement("contacts");

foreach (Customer c in customers)
    if (c.Country == "USA") {
        XmlElement e = doc.CreateElement("contact");
        XmlElement name = doc.CreateElement("name");
        name.InnerText = c.CompanyName;
        e.AppendChild(name);
        XmlElement phone = doc.CreateElement("phone");
        phone.InnerText = c.Phone;
        e.AppendChild(phone);
        contacts.AppendChild(e);
    }
doc.AppendChild(contacts);
```

```
<contacts>
  <contact>
    <name>Great Food</name>
    <phone>555-7123</phone>
  </contact>
  ...
</contacts>
```

LINQ to XML



```
XElement contacts = new
XElement("contacts",
from c in customers
where c.Country == "USA"
select new XElement("contact",
                    new XElement("name",
c.CompanyName),
                    new XElement("phone",
c.Phone)
)
);
```



## LINQ To XML - przykład

### Example:

Consider an XML file on disk that contains the data below:

```
<?xml version="1.0" encoding="utf-8" ?>
<people>
  <person age="15">
    <firstname>AAA</firstname> <lastname>XXX</lastname>
  </person>
  <person age="17">
    <firstname>ABB</firstname> <lastname>YYY</lastname>
  </person>
  <person age="24">
    <firstname>CCC</firstname> <lastname>ZZZ</lastname>
  </person>
</people>
```



## LINQ To XML – przykład

//Using LINQ Extension Methods against an XML File

```
XDocument people = XDocument.Load(@"C:\LINQToXML.xml");
```

//Casting to XElement

```
System.Collections.Generic.IEnumerable<XElement> xmlResult;
```

```
xmlResult = people.Descendants("person")
```

```
    .Where(p=>p.Element("firstname").Value.StartsWith("A"));
```

//Total count of records.

```
txtResultCount.Text = xmlResult.Count().ToString();
```

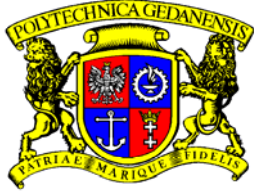
//Person First Name.

```
txtPersonFirstName.Text = xmlResult.First().FirstName;
```

//Person Last Name.

```
txtPersonLastName.text = xmlResult.First().LastName;
```

```
txtAvgAge.Text = people.Descendants("person").Average(p =>  
Convert.ToInt32(p.Attribute("age").Value));
```



## Benefits Of LINQ

- **Unified querying of objects, relational, XML**
- **Type checking and IntelliSense for queries**
- **SQL and XQuery-like power in C# and VB**
- **Extensibility model for languages / APIs**





# EDM Entity Data Model Basics

- The **Conceptual Model** is an **Entity Data Model (EDM)** schema that defines the entities and associations in the EDM.
  - The XML syntax that defines this model is called the **conceptual schema definition language (CSDL)**.
  - Entity types defined in CSDL each have a name, a key for uniquely identifying instances, and a set of properties
- Conceptual model based on **Entity-Relationship model**
  - **Entity types** are structures with a key, and can inherit from other entity types
  - An **Entity** is an instance of an entity type
  - Entities exist in **Entity Sets**
  - Entities are related to each other through **Associations**
    - They can be 1:1, 1:\* or \*:\*
- The **Storage Model** is a separate data model uses **store schema definition language (SSDL)** to describe the logical model for persistent data, usually stored in a relational database. The data types of properties declared in SSDL files are those of the storage model.
- A **mapping** specification uses **mapping specification language (MSL)** to connect the types declared in the **conceptual model** to the database metadata declared in the **storage model**.



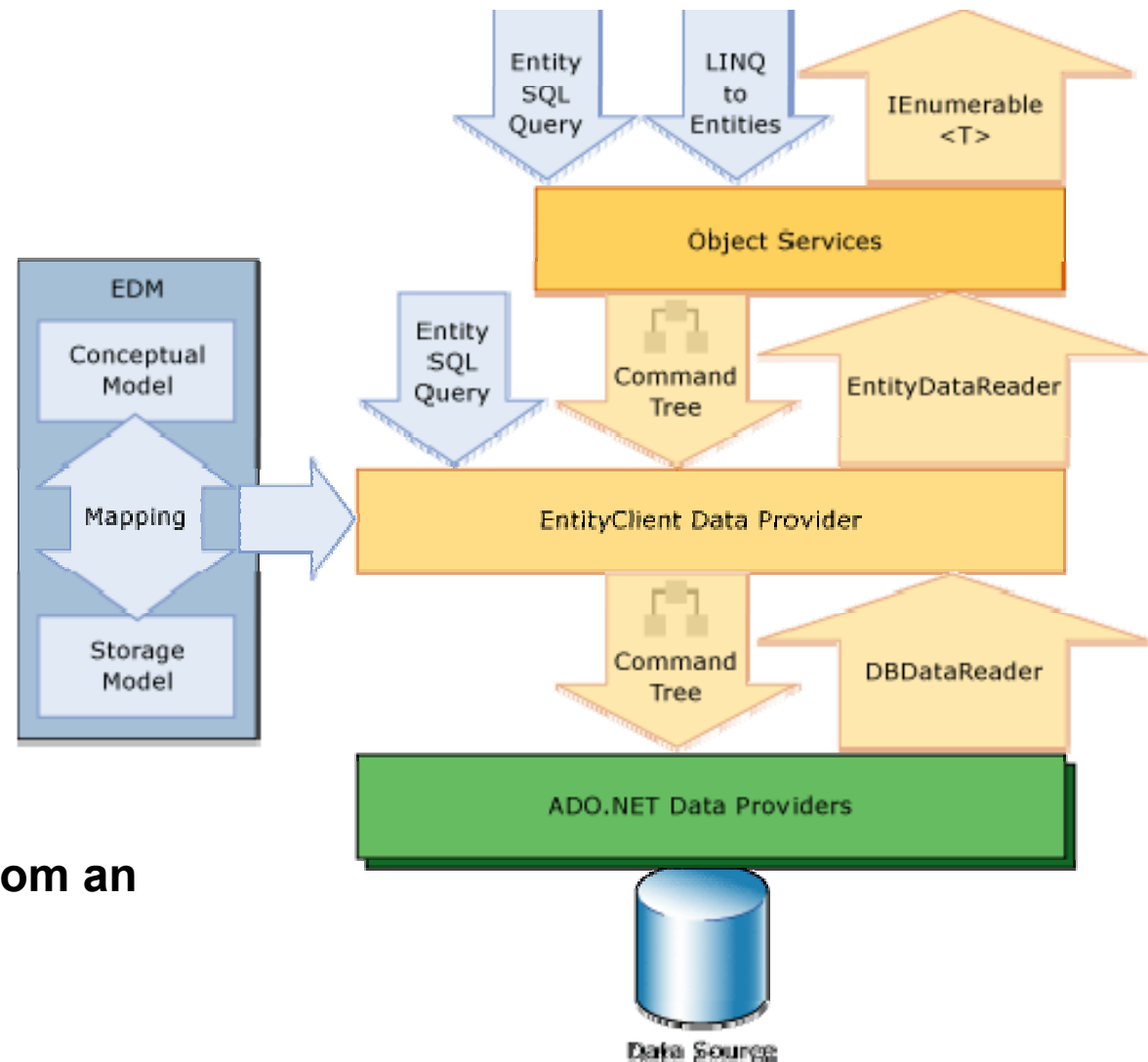
## Entity Framework architecture for accessing data

The Entity Framework generates a class derived from **ObjectContext** that represents the entity container in the conceptual model.

This **object context** provides the facilities for tracking changes and managing identities, concurrency, and relationships.

This class also exposes a **SaveChanges** method that writes inserts, updates, and deletes to the data source

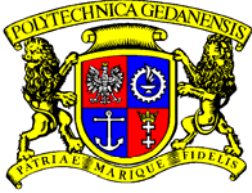
An Entity Data Model is derived from an existing database.





# Accessing and Changing Entity Data

- **Object Services** uses the EDM to translate object queries against entity types that are represented in the conceptual model into data source-specific queries
- To query an EDM and return objects:
  - **LINQ to Entities** - provides Language-Integrated Query (LINQ) support for querying **entity types** that are defined **in a conceptual model**
  - **Entity SQL** - a storage-independent dialect of SQL that works directly with **entities** in the **conceptual model** and that supports EDM features such as inheritance and relationships
  - **Query builder** methods - enables you to construct Entity SQL queries using LINQ-style query methods.



# Entity SQL (SELECT only)

```
// Object Services
using (NorthwindEntities db = new NorthwindEntities()) {

    // Entity SQL
    var q = db.CreateQuery<Products>("SELECT VALUE p FROM NorthwindEntities.Products AS p " +
        "WHERE p.UnitPrice > @price", new ObjectParameter("price", 60));

    foreach (var prod in q) {
        Console.WriteLine(prod.ProductName);
    }
}

// Entity Client
using (EntityConnection con = new EntityConnection("name=NorthwindEntities")) {
    con.Open();

    // Entity SQL
    EntityCommand cmd = new EntityCommand("SELECT p.ProductName FROM NorthwindEntities.Products"
        + " AS p WHERE p.UnitPrice > @price", con);

    cmd.Parameters.AddWithValue("price", 60);

    using (EntityDataReader r = cmd.ExecuteReader(CommandBehavior.SequentialAccess)) {
        while (r.Read()) {
            Console.WriteLine(r["ProductName"]);
        }
    }
}
```