LINQ to SQL: .NET Language-Integrated Query for Relational Data

Dinesh Kulkarni, Luca Bolognese, Matt Warren, Anders Hejlsberg, Kit George

March 2007

Applies to:  
   Visual Studio Code Name "Orcas"  
   .Net Framework 3.5

**Summary:** LINQ to SQL provides a runtime infrastructure for managing relational data as objects without losing the ability to query. Your application is free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically. (119 printed pages)

**Contents**

[Introduction](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic1)  
[A Quick Tour](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic2)  
   [Creating Entity Classes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic3)  
   [The DataContext](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic4)  
   [Defining Relationships](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic5)  
   [Querying Across Relationships](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic6)  
   [Modifying and Saving Entities](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic7)  
[Queries In-Depth](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic8)  
   [Query Execution](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic9)  
   [Object Identity](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic10)  
   [Relationships](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic11)  
   [Joins](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic12)  
   [Projections](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic13)  
   [Compiled Queries](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic14)  
   [SQL Translation](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic15)  
[The Entity Lifecycle](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic16)  
   [Tracking Changes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic17)  
   [Submitting Changes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic18)  
   [Simultaneous Changes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic19)  
   [Transactions](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic20)  
   [Stored Procedures](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic21)   
[Entity Classes In-Depth](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic22)  
   [Using Attributes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic23)  
   [Graph Consistency](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic24)  
   [Change Notifications](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic25)  
   [Inheritance](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic26)  
[Advanced Topics](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic27)  
   [Creating Databases](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic28)  
   [Interoperating with ADO.NET](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic29)  
   [Change Conflict Resolution](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic30)  
   [Stored Procedures Invocation](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic31)  
   [The Entity Class Generator Tool](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic32)  
   [Generator Tool DBML Reference](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic33)  
   [Multi-tier Entities](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic34)  
   [External Mapping](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic35)  
   [NET Framework Function Support and Notes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic36)  
   [Debugging Support](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic37)

**Introduction**

Most programs written today manipulate data in one way or another and often this data is stored in a relational database. Yet there is a huge divide between modern programming languages and databases in how they represent and manipulate information. This impedance mismatch is visible in multiple ways. Most notable is that programming languages access information in databases through APIs that require queries to be specified as text strings. These queries are significant portions of the program logic. Yet they are opaque to the language, unable to benefit from compile-time verification and design-time features like IntelliSense.

Of course, the differences go far deeper than that. How information is represented—the data model—is quite different between the two. Modern programming languages define information in the form of objects. Relational databases use rows. Objects have unique identity as each instance is physically different from another. Rows are identified by primary key values. Objects have references that identify and link instances together. Rows are left intentionally distinct requiring related rows to be tied together loosely using foreign keys. Objects stand alone, existing as long as they are still referenced by another object. Rows exist as elements of tables, vanishing as soon as they are removed.

It is no wonder that applications expected to bridge this gap are difficult to build and maintain. It would certainly simplify the equation to get rid of one side or the other. Yet relational databases provide critical infrastructure for long-term storage and query processing, and modern programming languages are indispensable for agile development and rich computation.

Until now, it has been the job of the application developer to resolve this mismatch in each application separately. The best solutions so far have been elaborate database abstraction layers that ferry the information between the applications domain-specific object models and the tabular representation of the database, reshaping and reformatting the data each way. Yet by obscuring the true data source, these solutions end up throwing away the most compelling feature of relational databases; the ability for the data to be queried.

LINQ to SQL, a component of Visual Studio Code Name "Orcas", provides a run-time infrastructure for managing relational data as objects without losing the ability to query. It does this by translating language-integrated queries into SQL for execution by the database, and then translating the tabular results back into objects you define. Your application is then free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically.

* LINQ to SQL is designed to be non-intrusive to your application.
  + It is possible to migrate current ADO.NET solutions to LINQ to SQL in a piecemeal fashion (sharing the same connections and transactions) since LINQ to SQL is simply another component in the ADO.NET family. LINQ to SQL also has extensive support for stored procedures, allowing reuse of the existing enterprise assets.
* LINQ to SQL applications are easy to get started.
  + Objects linked to relational data can be defined just like normal objects, only decorated with attributes to identify how properties correspond to columns. Of course, it is not even necessary to do this by hand. A design-time tool is provided to automate translating pre-existing relational database schemas into object definitions for you.

Together, the LINQ to SQL run-time infrastructure and design-time tools significantly reduce the workload for the database application developer. The following chapters provide an overview of how LINQ to SQL can be used to perform common database-related tasks. It is assumed that the reader is familiar with Language-Integrated Query and the standard query operators.

LINQ to SQL is language-agnostic. Any language built to provide Language-Integrated Query can use it to enable access to information stored in relational databases. The samples in this document are shown in both C# and Visual Basic; LINQ to SQL can be used with the LINQ-enabled version of the Visual Basic compiler as well.

**A Quick Tour**

The first step in building a LINQ to SQL application is declaring the object classes you will use to represent your application data. Let's walk through an example.

**Creating Entity Classes**

We will start with a simple class **Customer** and associate it with the customers table in the Northwind sample database. To do this, we need only apply a custom attribute to the top of the class declaration. LINQ to SQL defines the **Table** attribute for this purpose.

**C#**

[Table(Name="Customers")]

public class Customer

{

public string CustomerID;

public string City;

}

**Visual Basic**

<Table(Name:="Customers")> \_

Public Class Customer

Public CustomerID As String

Public City As String

End Class

The **Table** attribute has a **Name** property that you can use to specify the exact name of the database table. If no **Name** property is supplied, LINQ to SQL will assume the database table has the same name as the class. Only instances of classes declared as tables will be stored in the database. Instances of these types of classes are known as *entities*.The classes themselves are known as *entity classes*.

In addition to associating classes to tables you will need to denote each field or property you intend to associate with a database column. For this, LINQ to SQL defines the **Column** attribute.

**C#**

[Table(Name="Customers")]

public class Customer

{

[Column(IsPrimaryKey=true)]

public string CustomerID;

[Column]

public string City;

}

**Visual Basic**

<Table(Name:="Customers")> \_

Public Class Customer

<Column(IsPrimaryKey:=true)> \_

Public CustomerID As String

<Column> \_

Public City As String

End Class

The **Column** attribute has a variety of properties you can use to customize the exact mapping between your fields and the database columns. One property of note is the **Id** property. It tells LINQ to SQL that the database column is part of the primary key in the table.

As with the **Table** attribute, you only need to supply information in the **Column** attribute if it differs from what can be deduced from your field or property declaration. In this example, you need to tell LINQ to SQL that the **CustomerID** field is part of the primary key in the table, yet you don't have to specify the exact name or type.

Only fields and properties declared as columns will be persisted to or retrieved from the database. Others will be considered as transient parts of your application logic.

**The DataContext**

The **DataContext** is the main conduit by which you retrieve objects from the database and resubmit changes. You use it in the same way that you would use an ADO.NET Connection. In fact, the **DataContext** is initialized with a connection or connection string you supply. The purpose of the **DataContext** is to translate your requests for objects into SQL queries made against the database and then assemble objects out of the results. The **DataContext** enables *language-integrated query* by implementing the same operator pattern as the *standard query operators* such as **Where** and **Select**.

For example, you can use the **DataContext** to retrieve customer objects whose city is London as follows:

**C#**

// DataContext takes a connection string

DataContext db = new DataContext("c:\\northwind\\northwnd.mdf");

// Get a typed table to run queries

Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London

var q =

from c in Customers

where c.City == "London"

select c;

foreach (var cust in q)

Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City);

**Visual Basic**

' DataContext takes a connection string

Dim db As DataContext = New DataContext("c:\northwind\northwnd.mdf")

' Get a typed table to run queries

Dim Customers As Customers(Of Customer) = db.GetTable(Of Customer)()

' Query for customers from London

Dim londonCustomers = From customer in Customers \_

Where customer.City = "London" \_

Select customer

For Each cust in londonCustomers

Console.WriteLine("id = " & cust.CustomerID & ", City = " & cust.City)

Next

Each database table is represented as a **Table** collection, accessible via the **GetTable()** method using its entity class to identify it. It is recommended that you declare a strongly typed **DataContext** instead of relying on the basic **DataContext** class and the **GetTable()** method. A strongly typed **DataContext** declares all **Table** collections as members of the context.

**C#**

public partial class Northwind : DataContext

{

public Table<Customer> Customers;

public Table<Order> Orders;

public Northwind(string connection): base(connection) {}

}

**Visual Basic**

Partial Public Class Northwind

Inherits DataContext

Public Customers As Table(Of Customers)

Public Orders As Table(Of Orders)

Public Sub New(ByVal connection As String)

MyBase.New(connection)

End Sub

End Class

The query for customers from London can then be expressed more simply as:

**C#**

Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

var q =

from c in db.Customers

where c.City == "London"

select c;

foreach (var cust in q)

Console.WriteLine("id = {0}, City = {1}",cust.CustomerID, cust.City);

**Visual Basic**

Dim db = New Northwind("c:\northwind\northwnd.mdf")

Dim londonCustomers = From cust In db.Customers \_

Where cust.City = "London" \_

Select cust

For Each cust in londonCustomers

Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City)

Next

We will continue to use the strongly typed **Northwind** class for the remainder of the overview document.

**Defining Relationships**

Relationships in relational databases are typically modeled as foreign key values referring to primary keys in other tables. To navigate between them, you must explicitly bring the two tables together using a relational join operation. Objects, on the other hand, refer to each other using property references or collections of references navigated using "dot" notation. Obviously, dotting is simpler than joining, since you need not recall the explicit join condition each time you navigate.

For data relationships such as these that will always be the same, it becomes quite convenient to encode them as property references in your entity class. LINQ to SQL defines an **Association** attribute you can apply to a member used to represent a relationship. An *association relationship* is one like a foreign-key to primary-key relationship that is made by matching column values between tables.

**C#**

[Table(Name="Customers")]

public class Customer

{

[Column(Id=true)]

public string CustomerID;

...

private EntitySet<Order> \_Orders;

[Association(Storage="\_Orders", OtherKey="CustomerID")]

public EntitySet<Order> Orders {

get { return this.\_Orders; }

set { this.\_Orders.Assign(value); }

}

}

**Visual Basic**

<Table(Name:="Customers")> \_

Public Class Customer

<Column(Id:=true)> \_

Public CustomerID As String

...

Private \_Orders As EntitySet(Of Order)

<Association(Storage:="\_Orders", OtherKey:="CustomerID")> \_

Public Property Orders() As EntitySet(Of Order)

Get

Return Me.\_Orders

End Get

Set(ByVal value As EntitySet(Of Order))

End Set

End Property

End Class

The **Customer** class now has a property that declares the relationship between customers and their orders. The **Orders** property is of type **EntitySet** because the relationship is one-to-many. We use the **OtherKey** property in the **Association** attribute to describe how this association is done. It specifies the names of the properties in the related class to be compared with this one. There was also a **ThisKey** property we did not specify. Normally, we would use it to list the members on this side of the relationship. However, by omitting it we allow LINQ to SQL to infer them from the members that make up the primary key.

Notice how this is reversed in the definition for the **Order** class.

**C#**

[Table(Name="Orders")]

public class Order

{

[Column(Id=true)]

public int OrderID;

[Column]

public string CustomerID;

private EntityRef<Customer> \_Customer;

[Association(Storage="\_Customer", ThisKey="CustomerID")]

public Customer Customer {

get { return this.\_Customer.Entity; }

set { this.\_Customer.Entity = value; }

}

}

**Visual Basic**

<Table(Name:="Orders")> \_

Public Class Order

<Column(Id:=true)> \_

Public OrderID As String

<Column> \_

Public CustomerID As String

Private \_Customer As EntityRef(Of Customer)

<Association(Storage:="\_Customer", ThisKey:="CustomerID")> \_

Public Property Customer() As Customer

Get

Return Me.\_Customer.Entity

End Get

Set(ByVal value As Customer)

Me.\_Customers.Entity = value

End Set

End Property

End Class

The **Order** class uses the **EntityRef** type to describe the relationship back to the customer. The use of the **EntityRef** class is required to support *deferred loading* (discussed later). The **Association** attribute for the **Customer** property specifies the **ThisKey** property since the non-inferable members are now on this side of the relationship.

Also take a look at the **Storage** property. It tells LINQ to SQL which private member is used to hold the value of the property. This allows LINQ to SQL to bypass your public property accessors when it stores and retrieves their value. This is essential if you want LINQ to SQL to avoid any custom business logic written into your accessors. If the storage property is not specified, the public accessors will be used instead. You may use the **Storage** property with **Column** attributes as well.

Once you introduce relationships in your entity classes, the amount of code you need to write grows as you introduce support for notifications and graph consistency. Fortunately, there is a tool (described later) that can be used to generate all the necessary definitions as partial classes, allowing you to use a mix of generated code and custom business logic.

For the rest of this document, we assume the tool has been used to generate a complete **Northwind** data context and all entity classes.

**Querying Across Relationships**

Now that you have relationships, you can use them when you write queries simply by referring to the relationship properties defined in your class.

**C#**

var q =

from c in db.Customers

from o in c.Orders

where c.City == "London"

select new { c, o };

**Visual Basic**

Dim londonCustOrders = From cust In db.Customers, ord In cust.Orders \_

Where cust.City = "London" \_

Select Customer = cust, Order = ord

The above query uses the **Orders** property to form the cross product between customers and orders, producing a new sequence of **Customer** and **Order** pairs.

It's also possible to do the reverse.

**C#**

var q =

from o in db.Orders

where o.Customer.City == "London"

select new { c = o.Customer, o };

**Visual Basic**

Dim londonCustOrders = From ord In db.Orders \_

Where ord.Customer.City = "London" \_

Select Customer = ord.Customer, Order = ord

In this example, the orders are queried and the **Customer** relationship is used to access information on the associated **Customer** object.

**Modifying and Saving Entities**

Few applications are built with only query in mind. Data must be created and modified, too. LINQ to SQL is designed to offer maximum flexibility in manipulating and persisting changes made to your objects. As soon as entity objects are available—either by retrieving them through a query or constructing them anew—you may manipulate them as normal objects in your application, changing their values or adding and removing them from collections as you see fit. LINQ to SQL tracks all your changes and is ready to transmit them back to the database as soon as you are done.

The example below uses the **Customer** and **Order** classes generated by a tool from the metadata of the entire Northwind sample database. The class definitions have not been shown for brevity.

**C#**

Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// Query for a specific customer

string id = "ALFKI";

var cust = db.Customers.Single(c => c.CustomerID == id);

// Change the name of the contact

cust.ContactName = "New Contact";

// Create and add a new Order to Orders collection

Order ord = new Order { OrderDate = DateTime.Now };

cust.Orders.Add(ord);

// Ask the DataContext to save all the changes

db.SubmitChanges();

**Visual Basic**

Dim db As New Northwind("c:\northwind\northwnd.mdf")

' Query for a specific customer

Dim id As String = "ALFKI"

Dim targetCustomer = (From cust In db.Customers \_

Where cust.CustomerID = id).First

' Change the name of the contact

targetCustomer.ContactName = "New Contact"

' Create and add a new Order to Orders collection

Dim id = New Order With { .OrderDate = DateTime.Now }

targetCustomer.Orders.Add(ord)

' Ask the DataContext to save all the changes

db.SubmitChanges()

When **SubmitChanges()** is called, LINQ to SQL automatically generates and executes SQL commands in order to transmit the changes back to the database. It is also possible to override this behavior with custom logic. The custom logic may call a database stored procedure.

**Queries In-Depth**

LINQ to SQL provides an implementation of the standard query operators for objects associated with tables in a relational database. This chapter describes the LINQ to SQL-specific aspects of queries.

**Query Execution**

Whether you write a query as a high-level *query expression* or build one out of the individual operators, the query that you write is not an imperative statement executed immediately. It is a description. For example, in the declaration below the local variable **q** refers to the description of the query not the result of executing it.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c;

foreach (Customer c in q)

Console.WriteLine(c.CompanyName);

**Visual Basic**

Dim londonCustomers = From cust In db.Customers \_

where cust.City = "London"

For Each cust In londonCustomers

Console.WriteLine(cust.CompanyName)

Next

The actual type of **q** in this instance is **IQueryable<Customer>**. It's not until the application attempts to enumerate the contents of the query that it actually executes. In this example the **foreach** statement causes the execution to occur.

An **IQueryable** object is similar to an ADO.NET command object. Having one in hand does not imply that a query was executed. A command object holds onto a string that describes a query. Likewise, an **IQueryable** object holds onto a description of a query encoded as a data structure known as an **Expression**. A command object has an **ExecuteReader()** method that causes execution, returning results as a **DataReader**. An **IQueryable** object has a **GetEnumerator()** method that causes the execution, returning results as an **IEnumerator<Customer>**.

Therefore, it follows that if a query is enumerated twice it will be executed twice.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c;

// Execute first time

foreach (Customer c in q)

Console.WriteLine(c.CompanyName);

// Execute second time

foreach (Customer c in q)

Console.WriteLine(c.CompanyName);

**Visual Basic**

Dim londonCustomers = From cust In db.Customers \_

where cust.City = "London"

' Execute first time

For Each cust In londonCustomers

Console.WriteLine(cust.CompanyName)

Next

' Execute second time

For Each cust In londonCustomers

Console.WriteLine(cust.CustomerID)

Next

This behavior is known as *deferred execution*. Just like with an ADO.NET command object it is possible to hold onto a query and re-execute it.

Of course, application writers often need to be very explicit about where and when a query is executed. It would be unexpected if an application were to execute a query multiple times simply because it needed to examine the results more than once. For example, you may want to bind the results of a query to something like a DataGrid. The control may enumerate the results each time it paints on the screen.

To avoid executing multiple times convert the results into any number of standard collection classes. It is easy to convert the results into a list or array using the standard query operators **ToList()** or **ToArray()**.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c;

// Execute once using ToList() or ToArray()

var list = q.ToList();

foreach (Customer c in list)

Console.WriteLine(c.CompanyName);

foreach (Customer c in list)

Console.WriteLine(c.CompanyName);

**Visual Basic**

Dim londonCustomers = From cust In db.Customers \_

where cust.City = "London"

' Execute once using ToList() or ToArray()

Dim londonCustList = londonCustomers.ToList()

' Neither of these iterations re-executes the query

For Each cust In londonCustList

Console.WriteLine(cust.CompanyName)

Next

For Each cust In londonCustList

Console.WriteLine(cust.CompanyName)

Next

One benefit of deferred execution is that queries may be piecewise constructed with execution only occurring when the construction is complete. You can start out composing a portion of a query, assigning it to a local variable and then sometime later continue applying more operators to it.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c;

if (orderByLocation) {

q =

from c in q

orderby c.Country, c.City

select c;

}

else if (orderByName) {

q =

from c in q

orderby c.ContactName

select c;

}

foreach (Customer c in q)

Console.WriteLine(c.CompanyName);

**Visual Basic**

Dim londonCustomers = From cust In db.Customers \_

where cust.City = "London"

if orderByLocation Then

londonCustomers = From cust in londonCustomers \_

Order By cust.Country, cust.City

Else If orderByName Then

londonCustomers = From cust in londonCustomers \_

Order By cust.ContactName

End If

For Each cust In londonCustList

Console.WriteLine(cust.CompanyName)

Next

In this example, **q** starts out as a query for all customers in London. Later on it changes into an ordered query depending on application state. By deferring execution the query can be constructed to suit the exact needs of the application without requiring risky string manipulation.

**Object Identity**

Objects in the runtime have unique identity. If two variables refer to the same object, they are actually referring to the same object instance. Because of this, changes made via a path through one variable are immediately visible through the other. Rows in a relational database table do not have unique identity. However, they do have a primary key and that primary key may be unique, meaning no two rows may share the same key. Yet this only constrains the contents of the database table. Therefore, as long as we only interact with the data through remote commands, it amounts to about the same thing.

However, this is rarely the case. Most often data is brought out of the database and into a different tier where an application manipulates it. Clearly, this is the model that LINQ to SQL is designed to support. When the data is brought out of the database as rows, there is no expectation that two rows representing the same data actually correspond to the same row instances. If you query for a specific customer twice, you get two rows of data, each containing the same information.

Yet with objects, you expect something quite different. You expect that if you ask the **DataContext** for the same information again, it will in fact give you back the same object instance. You expect this because objects have special meaning for your application and you expect them to behave like normal objects. You designed them as hierarchies or graphs and you certainly expect to retrieve them as such, without hordes of replicated instances merely because you asked for the same thing twice.

Because of this, the **DataContext** manages object identity. Whenever a new row is retrieved from the database, it is logged in an identity table by its primary key and a new object is created. Whenever that same row is retrieved again, the original object instance is handed back to the application. In this way, the **DataContext** translates the databases concept of identity (keys) into the languages concept (instances). The application only ever sees the object in the state that it was first retrieved. The new data, if different, is thrown away.

You might be puzzled by this, since why would any application throw data away? As it turns out this is how LINQ to SQL manages integrity of the local objects and is able to support optimistic updates. Since the only changes that occur after the object is initially created are those made by the application, the intent of the application is clear. If changes by an outside party have occurred in the interim they will be identified at the time **SubmitChanges()** is called. More of this is explained in the [Simultaneous Changes](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic19) section.

Note that, in the case that the database contains a table without a primary key, LINQ to SQL allows queries to be submitted over the table, but it doesn't allow updates. This is because the framework cannot identify which row to update given the lack of a unique key.

Of course, if the object requested by the query is easily identifiable by its primary key as one already retrieved no query is executed at all. The identity table acts as a cache storing all previously retrieved objects.

**Relationships**

As we saw in the quick tour, references to other objects or collections of other objects in your class definitions directly correspond to foreign-key relationships in the database. You can use these relationships when you query by simply using dot notation to access the relationship properties, navigating from one object to another. These access operations translate to more complicated joins or correlated sub-queries in the equivalent SQL, allowing you to walk through your object graph during a query. For example, the following query navigates from orders to customers as a way to restrict the results to only those orders for customers located in London.

**C#**

var q =

from o in db.Orders

where o.Customer.City == "London"

select o;

**Visual Basic**

Dim londonOrders = From ord In db.Orders \_

where ord.Customer.City = "London"

If relationship properties did not exist you would have to write them out manually as joins just as you would do in a SQL query.

**C#**

var q =

from c in db.Customers

join o in db.Orders on c.CustomerID equals o.CustomerID

where c.City == "London"

select o;

**Visual Basic**

Dim londonOrders = From cust In db.Customers \_

Join ord In db.Orders \_

On cust.CustomerID Equals ord.CustomerID \_

Where ord.Customer.City = "London" \_

Select ord

The relationship property allows you to define this particular relationship once enabling the use of the more convenient dot syntax. However, this is not the reason why relationship properties exist. They exist because we tend to define our domain-specific object models as hierarchies or graphs. The objects we choose to program against have references to other objects. It's only a happy coincidence that since object-to-object relationships correspond to foreign key style relationships in databases that property access leads to a convenient way to write joins.

Therefore, the existence of relationship properties is more important on the results side of a query than as part of the query itself. Once you have your hands on a particular customer, its class definition tells you that customers have orders. So when you look into the **Orders** property of a particular customer you expect to see the collection populated with all the customer's orders, since that is in fact the contract you declared by defining the classes this way. You expect to see the orders there even if you did not particularly ask for orders up front. You expect your object model to maintain an illusion that it is an in-memory extension of the database, with related objects immediately available.

LINQ to SQL implements a technique called *deferred loading* in order to help maintain this illusion. When you query for an object you actually only retrieve the objects you asked for. The related objects are not automatically fetched at the same time. However, the fact that the related objects are not already loaded is not observable since as soon as you attempt to access them a request goes out to retrieve them.

**C#**

var q =

from o in db.Orders

where o.ShipVia == 3

select o;

foreach (Order o in q) {

if (o.Freight > 200)

SendCustomerNotification(o.Customer);

ProcessOrder(o);

}

**Visual Basic**

Dim shippedOrders = From ord In db.Orders \_

where ord.ShipVia = 3

For Each ord In shippedOrders

If ord.Freight > 200 Then

SendCustomerNotification(ord.Customer)

ProcessOrder(ord)

End If

Next

For example, you may want to query for a particular set of orders and then only occasionally send an email notification to particular customers. You would not necessary need to retrieve all customer data up front with every order. Deferred loading allows you to defer the cost of retrieving extra information until you absolutely have to.

Of course, the opposite might also be true. You might have an application that needs to look at customer and order data at the same time. You know you need both sets of data. You know your application is going to drill down through each customer's orders as soon as you get them. It would be unfortunate to fire off individual queries for orders for every customer. What you really want to happen is to have the order data retrieved together with the customers.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c;

foreach (Customer c in q) {

foreach (Order o in c.Orders) {

ProcessCustomerOrder(o);

}

}

**Visual Basic**

Dim londonCustomers = From cust In db.Customer \_

Where cust.City = "London"

For Each cust In londonCustomers

For Each ord In cust.Orders

ProcessCustomerOrder(ord)

End If

Next

Certainly, you can always find a way to join customers and orders together in a query by forming the cross product and retrieving all the relative bits of data as one big projection. But then the results would not be entities. Entities are objects with identity that you can modify while the results would be projections that cannot be changed and persisted. Worse, you would be retrieving a huge amount of redundant data as each customer repeats for each order in the flattened join output.

What you really need is a way to retrieve a set of related objects at the same time—a delineated portion of a graph so you would never be retrieving any more or any less than was necessary for your intended use.

LINQ to SQL allows you to request *immediate loading* of a region of your object model for just this reason. It does this by allowing the specification of a **DataShape** for a **DataContext**. The **DataShape** class is used to instruct the framework about which objects to retrieve when a particular type is retrieved. This is accomplished by using the **LoadWith** method as in the following:

**C#**

DataShape ds = new DataShape();

ds.LoadWith<Customer>(c => c.Orders);

db.Shape = ds;

var q =

from c in db.Customers

where c.City == "London"

select c;

**Visual Basic**

Dim ds As DataShape = New DataShape()

ds.LoadWith(Of Customer)(Function(c As Customer) c.Orders)

db.Shape = ds

Dim londonCustomers = From cust In db.Customers \_

Where cust.City = "London" \_

Select cust

In the previous query, all the **Orders** for all the **Customers** who live in London are retrieved when the query is executed, so that successive access to the **Orders** property on a **Customer** object doesn't trigger a database query.

The **DataShape** class can also be used to specify sub-queries that are applied to a relationship navigation. For example, if you want to retrieve just the **Orders** that have been shipped today, you can use the **AssociateWith** method on the **DataShape** as in the following:

**C#**

DataShape ds = new DataShape();

ds.AssociateWith<Customer>(

c => c.Orders.Where(p => p.ShippedDate != DateTime.Today));

db.Shape = ds;

var q =

from c in db.Customers

where c.City == "London"

select c;

foreach(Customer c in q) {

foreach(Order o in c.Orders) {}

}

**Visual Basic**

Dim ds As DataShape = New DataShape()

ds.AssociateWith(Of Customer)( \_

Function(cust As Customer) From cust In db.Customers \_

Where order.ShippedDate <> Today \_

Select cust)

db.Shape = ds

Dim londonCustomers = From cust In db.Customers \_

Where cust.City = "London" \_

Select cust

For Each cust in londonCustomers

For Each ord In cust.Orders …

Next

Next

In the previous code, the inner **foreach** statement iterates just over the **Orders** that have been shipped today, because just such orders have been retrieved from the database.

It is important to notice two facts about the **DataShape** class:

1. After assigning a **DataShape** to a **DataContext**, the **DataShape** cannot be modified. Any **LoadWith** or **AssociateWith** method call on such a **DataShape** will return an error at run time.
2. It is impossible to create cycles by using **LoadWith** or **AssociateWith**. For example, the following generates an error at run time:

**C#**

DataShape ds = new DataShape();

ds.AssociateWith<Customer>(

c=>c.Orders.Where(o=> o.Customer.Orders.Count() < 35);

**Visual Basic**

Dim ds As DataShape = New DataShape()

ds.AssociateWith(Of Customer)( \_

Function(cust As Customer) From ord In cust.Orders \_

Where ord.Customer.Orders.Count() < 35)

**Joins**

Most queries against object models heavily rely on navigating object references in the object model. However, there are interesting "relationships" between entities that may not be captured in the object model as references. For example **Customer.Orders** is a useful relationship based on foreign key relationships in the Northwind database. However, Suppliers and Customers in the same City or Country is an *ad hoc* relationship that is not based on a foreign key relationship and may not be captured in the object model. Joins provide an additional mechanism to handle such relationships. LINQ to SQL supports the new join operators introduced in LINQ.

Consider the following problem—find suppliers and customers based in the same city. The following query returns supplier and customer company names and the common city as a flattened result. This is the equivalent of the inner equi-join in relational databases:

**C#**

var q =

from s in db.Suppliers

join c in db.Customers on s.City equals c.City

select new {

Supplier = s.CompanyName,

Customer = c.CompanyName,

City = c.City

};

**Visual Basic**

Dim customerSuppliers = From sup In db.Suppliers \_

Join cust In db.Customers \_

On sup.City Equals cust.City \_

Select Supplier = sup.CompanyName, \_

CustomerName = cust.CompanyName, \_

City = cust.City

The above query eliminates suppliers that are not in the same city as a certain customer. However, there are times when we don't want to eliminate one of the entities in an *ad hoc* relationship. The following query lists all suppliers with groups of customers for each of the suppliers. If a particular supplier does not have any customer in the same city, the result is an empty collection of customers corresponding to that supplier. Note that the results are not flat—each supplier has an associated *collection*. Effectively, this provides group join—it joins two sequences and groups elements of the second sequence by the elements of the first sequence.

**C#**

var q =

from s in db.Suppliers

join c in db.Customers on s.City equals c.City into scusts

select new { s, scusts };

**Visual Basic**

Dim customerSuppliers = From sup In db.Suppliers \_

Group Join cust In db.Customers \_

On sup.City Equals cust.City \_

Into supCusts \_

Select Supplier = sup, \_

Customers = supCusts

Group join can be extended to multiple collections as well. The following query extends the above query by listing employees that are in the same city as the supplier. Here, the result shows a supplier with (possibly empty) collections of customers and employees.

**C#**

var q =

from s in db.Suppliers

join c in db.Customers on s.City equals c.City into scusts

join e in db.Employees on s.City equals e.City into semps

select new { s, scusts, semps };

**Visual Basic**

Dim customerSuppliers = From sup In db.Suppliers \_

Group Join cust In db.Customers \_

On sup.City Equals cust.City \_

Into supCusts \_

Group Join emp In db.Employees \_

On sup.City Equals emp.City \_

Into supEmps \_

Select Supplier = sup, \_

Customers = supCusts, Employees = supEmps

The results of a group join can also be flattened. The results of flattening the group join between suppliers and customers are multiple entries for suppliers with multiple customers in their city—one per customer. Empty collections are replaced with nulls. This is equivalent to a left outer equi-join in relational databases.

**C#**

var q =

from s in db.Suppliers

join c in db.Customers on s.City equals c.City into sc

from x in sc.DefaultIfEmpty()

select new {

Supplier = s.CompanyName,

Customer = x.CompanyName,

City = x.City

};

**Visual Basic**

Dim customerSuppliers = From sup In db.Suppliers \_

Group Join cust In db.Customers \_

On sup.City Equals cust.City \_

Into supCusts \_

Select Supplier = sup, \_

CustomerName = supCusts.CompanyName, sup.City

The signatures for underlying join operators are defined in the standard query operators document. Only equi-joins are supported and the two operands of **equals** must have the same type.

**Projections**

So far, we have only looked at queries for retrieving *entities*—objects directly associated with database tables. We need not constrain ourselves to just this. The beauty of a query language is that you can retrieve information in any form you want. You will not be able to take advantage of automatic change tracking or identity management when you do so. However, you can get just the data you want.

For example, you may simply need to know the company names of all customers in London. If this is the case there is no particular reason to retrieve entire customer objects merely to pick out names. You can project out the names as part of the query.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select c.CompanyName;

**Visual Basic**

Dim londonCustomerNames = From cust In db.Customer \_

Where cust.City = "London" \_

Select cust.CompanyName

In this case, **q** becomes a query that retrieves a sequence of strings.

If you want to get back more than just a single name, but not enough to justify fetching the entire customer object, you can specify any subset you want by constructing the results as part of your query.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new { c.CompanyName, c.Phone };

**Visual Basic**

Dim londonCustomerInfo = From cust In db.Customer \_

Where cust.City = "London" \_

Select cust.CompanyName, cust.Phone

This example uses an *anonymous object initializer* to create a structure that holds both the company name and phone number. You may not know what to call the type, but with *implicitly typed local variable declaration* in the language you do not necessarily need to.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new { c.CompanyName, c.Phone };

foreach(var c in q)

Console.WriteLine("{0}, {1}", c.CompanyName, c.Phone);

**Visual Basic**

Dim londonCustomerInfo = From cust In db.Customer \_

Where cust.City = "London" \_

Select cust.CompanyName, cust.Phone

For Each cust In londonCustomerInfo

Console.WriteLine(cust.CompanyName & ", " & cust.Phone)

Next

If you are consuming the data immediately, *anonymous types* make a good alternative to explicitly defining classes to hold your query results.

You can also form cross products of entire objects, though you might rarely have a reason to do so.

**C#**

var q =

from c in db.Customers

from o in c.Orders

where c.City == "London"

select new { c, o };

**Visual Basic**

Dim londonOrders = From cust In db.Customer, \_

ord In db.Orders \_

Where cust.City = "London" \_

Select Customer = cust, Order = ord

This query constructs a sequence of pairs of customer and order objects.

It's also possible to make projections at any stage of the query. You can project data into newly constructed objects and then refer to those objects' members in subsequent query operations.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new {Name = c.ContactName, c.Phone} into x

orderby x.Name

select x;

**Visual Basic**

Dim londonItems = From cust In db.Customer \_

Where cust.City = "London" \_

Select Name = cust.ContactName, cust.Phone \_

Order By Name

Be wary of using parameterized constructors at this stage, though. It is technically valid to do so, yet it is impossible for LINQ to SQL to track how constructor usage affects member state without understanding the actual code inside the constructor.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new MyType(c.ContactName, c.Phone) into x

orderby x.Name

select x;

**Visual Basic**

Dim londonItems = From cust In db.Customer \_

Where cust.City = "London" \_

Select MyType = New MyType(cust.ContactName, cust.Phone) \_

Order By MyType.Name

Because LINQ to SQL attempts to translate the query into pure relational SQL locally defined object types are not available on the server to actually construct. All object construction is actually postponed until after the data is retrieved back from the database. In place of actual constructors, the generated SQL uses normal SQL column projection. Since it is not possible for the query translator to understand what is happening during a constructor call, it is unable to establish a meaning for the **Name** field of **MyType**.

Instead, the best practice is to always use *object initializers* to encode projections.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new MyType { Name = c.ContactName, HomePhone = c.Phone } into x

orderby x.Name

select x;

**Visual Basic**

Dim londonCustomers = From cust In db.Customer \_

Where cust.City = "London" \_

Select Contact = New With {.Name = cust.ContactName, \_

.Phone = cust.Phone} \_

Order By Contact.Name

The only safe place to use a parameterized constructor is in the final projection of a query.

**C#**

var e =

new XElement("results",

from c in db.Customers

where c.City == "London"

select new XElement("customer",

new XElement("name", c.ContactName),

new XElement("phone", c.Phone)

)

);

**Visual Basic**

Dim x = <results>

<%= From cust In db.Customers \_

Where cust.City = "London" \_

Select <customer>

<name><%= cust.ContactName %></name>

<phone><%= cust.Phone %></phone>

</customer>

%>

</results>

You can even use elaborate nesting of object constructors if you desire, like this example that constructs XML directly out of the result of a query. It works as long as it's the last projection of the query.

Still, even if constructor calls are understood, calls to local methods may not be. If your final projection requires invocation of local methods, it is unlikely that LINQ to SQL will be able to oblige. Method calls that do not have a known translation into SQL cannot be used as part of the query. One exception to this rule is method calls that have no arguments dependent on query variables. These are not considered part of the translated query and instead are treated as parameters.

Still elaborate projections (transformations) may require local procedural logic to implement. For you to use your own local methods in a final projection you will need to project twice. The first projection extracts all the data values you'll need to reference and the second projection performs the transformation. In between these two projections is a call to the **AsEnumerable()** operator that shifts processing at that point from a LINQ to SQL query into a locally executed one.

**C#**

var q =

from c in db.Customers

where c.City == "London"

select new { c.ContactName, c.Phone };

var q2 =

from c in q.AsEnumerable()

select new MyType {

Name = DoNameProcessing(c.ContactName),

Phone = DoPhoneProcessing(c.Phone)

};

**Visual Basic**

Dim londonCustomers = From cust In db.Customer \_

Where cust.City = "London" \_

Select cust.ContactName, cust.Phone

Dim processedCustomers = From cust In londonCustomers.AsEnumerable() \_

Select Contact = New With { \_

.Name = DoNameProcessing(cust.ContactName), \_

.Phone = DoPhoneProcessing(cust.Phone)}

**Note**   The **AsEnumerable()** operator, unlike **ToList()** and **ToArray()**, does not cause execution of the query. It is still deferred. The **AsEnumerable()** operator merely changes the static typing of the query, turning a **IQueryable<T>** (**IQueryable (ofT)** in Visual Basic) into an **IEnumerable<T>** (**IEnumerable (ofT)** in Visual Basic), tricking the compiler into treating the rest of the query as locally executed.

**Compiled Queries**

It is common in many applications to execute structurally similar queries many times. In such cases, it is possible to increase performance by compiling the query once and executing it several times in the application with different parameters. This result is obtained in LINQ to SQL by using the **CompiledQuery** class. The following code shows how to define a compiled query:

**C#**

static class Queries

{

public static Func<Northwind, string, IQueryable<Customer>>

CustomersByCity = CompiledQuery.Compile((Northwind db, string city) =>

from c in db.Customers where c.City == city select c);

}

**Visual Basic**

Class Queries

public Shared Function(Of Northwind, String, IQueryable(Of Customer)) \_ CustomersByCity = CompiledQuery.Compile( \_

Function(db As Northwind, city As String) \_

From cust In db.Customers Where cust.City = city)

End Class

The **Compile** method returns a delegate that can be cached and executed afterward several times by just changing the input parameters. The following code shows an example of this:

**C#**

public IEnumerable<Customer> GetCustomersByCity(string city) {

Northwind db = new Northwind();

return Queries.CustomersByCity(myDb, city);

}

**Visual Basic**

Public Function GetCustomersByCity(city As String) \_

As IEnumerable(Of Customer)

Dim db As Northwind = New Northwind()

Return Queries.CustomersByCity(myDb, city)

End Function

**SQL Translation**

LINQ to SQL does not actually execute queries; the relational database does. LINQ to SQL translates the queries you wrote into equivalent SQL queries and sends them to the server for processing. Because execution is deferred, LINQ to SQL is able to examine your entire query even if assembled from multiple parts.

Since the relational database server is not actually executing IL (aside from the CLR integration in SQL Server 2005); the queries are not transmitted to the server as IL. They are in fact transmitted as parameterized SQL queries in text form.

Of course, SQL—even T-SQL with CLR integration—is incapable of executing the variety of methods that are locally available to your program. Therefore the queries you write must be translated into equivalent operations and functions that are available inside the SQL environment.

Most methods and operators on .Net Framework built-in types have direct translations into SQL. Some can be produced out of the functions that are available. The ones that cannot be translated are disallowed, generating run-time exceptions if you try to use them. There is a section later in the document that details the framework methods that are implemented to translate into SQL.

**The Entity Lifecycle**

LINQ to SQL is more than just an implementation of the standard query operators for relational databases. In addition to translating queries, it is a service that manages your objects throughout their lifetime, aiding you in maintaining the integrity of your data and automating the process of translating your modifications back into the store.

In a typical scenario, objects are retrieved through one or more queries and then manipulated in some way or another until the application is ready to send the changes back to the server. This process may repeat a number of times until the application no longer has use for this information. At that point, the objects are reclaimed by the runtime just like normal objects. The data, however, remains in the database. Even after being erased from their run-time existence, objects representing the same data can still be retrieved. In this sense, the object's true lifetime exists beyond any single run-time manifestation.

The focus of this chapter is the *entity lifecycle* where a cycle refers to the time span of a single manifestation of an entity object within a particular run-time context. The cycle starts when the **DataContext** becomes aware of a new instance and ends when the object or **DataContext** is no longer needed.

**Tracking Changes**

After entities are retrieved from the database, you are free to manipulate them as you like. They are your objects; use them as you will. As you do this, LINQ to SQL tracks changes so that it can persist them into the database when **SubmitChanges()** is called.

LINQ to SQL starts tracking your entities the moment they are retrieved from the database, before you ever lay your hands on them. Indeed, the *identity management service* discussed earlier has already kicked in as well. Change tracking costs very little in additional overhead until you actually start making changes.

**C#**

Customer cust = db.Customers.Single(c => c.CustomerID == "ALFKI");

cust.CompanyName = "Dr. Frogg's Croakers";

**Visual Basic**

' Query for a specific customer

Dim id As String = "ALFKI"

Dim targetCustomer = (From cust In db.Customers \_

Where cust.CustomerID = id).First

targetCustomer.CompanyName = "Dr. Frogg's Croakers"

As soon as the **CompanyName** is assigned in the example above, LINQ to SQL becomes aware of the change and is able to record it. The original values of all data members are retained by the *change tracking service*.

The change tracking service also records all manipulations of relationship properties. You use relationship properties to establish the links between your entities, even though they may be linked by key values in the database. There is no need to directly modify the members associated with the key columns. LINQ to SQL automatically synchronizes them for you before the changes are submitted.

**C#**

Customer cust1 = db.Customers.Single(c => c.CustomerID == custId1);

foreach (Order o in db.Orders.Where(o => o.CustomerID == custId2)) {

o.Customer = cust1;

}

**Visual Basic**

Dim targetCustomer = (From cust In db.Customers \_

Where cust.CustomerID = custId1).First

For Each ord In (From o In db.Orders \_

Where o.CustomerID = custId2)

o.Customer = targetCustomer

Next

You can move orders from one customer to another by simply making an assignment to their **Customer** property. Since the relationship exists between the customer and the order, you can change the relationship by modifying either side. You could have just as easily removed them from the **Orders** collection of **cust2** and added them to the orders collection of **cust1**, as shown below.

**C#**

Customer cust1 = db.Customers.Single(c => c.CustomerID == custId1);

Customer cust2 = db.Customers.Single(c => c.CustomerID == custId2);

// Pick some order

Order o = cust2.Orders[0];

// Remove from one, add to the other

cust2.Orders.Remove(o);

cust1.Orders.Add(o);

// Displays 'true'

Console.WriteLine(o.Customer == cust1);

**Visual Basic**

Dim targetCustomer1 = (From cust In db.Customers \_

Where cust.CustomerID = custId1).First

Dim targetCustomer2 = (From cust In db.Customers \_

Where cust.CustomerID = custId1).First

' Pick some order

Dim o As Order = targetCustomer2.Orders(0)

' Remove from one, add to the other

targetCustomer2.Orders.Remove(o)

targetCustomer1.Orders.Add(o)

' Displays 'True'

MsgBox(o.Customer = targetCustomer1)

Of course, if you assign a relationship the value of **null**, you are in fact getting rid of the relationship completely. Assigning a **Customer** property of an order to **null** actually removes the order from the customer's list.

**C#**

Customer cust = db.Customers.Single(c => c.CustomerID == custId1);

// Pick some order

Order o = cust.Orders[0];

// Assign null value

o.Customer = null;

// Displays 'false'

Console.WriteLine(cust.Orders.Contains(o));

**Visual Basic**

Dim targetCustomer = (From cust In db.Customers \_

Where cust.CustomerID = custId1).First

' Pick some order

Dim o As Order = targetCustomer.Orders(0)

' Assign null value

o.Customer = Nothing

' Displays 'False'

Msgbox(targetCustomer.Orders.Contains(o))

Automatic updating of both sides of a relationship is essential for maintaining consistency of your object graph. Unlike normal objects, relationships between data are often bidirectional. LINQ to SQL allows you to use properties to represent relationships. However, it does not offer a service to automatically keep these bidirectional properties in sync. This is a level of service that must be baked directly into your class definitions. Entity classes generated using the code generation tool have this capability. In the next chapter, we will show you how to do this to your own handwritten classes.

It is important to note, however, that removing a relationship does not imply that an object has been deleted from the database. Remember, the lifetime of the underlying data persists in the database until the row has been deleted from the table. The only way to actually delete an object is to remove it from its **Table** collection.

**C#**

Customer cust = db.Customers.Single(c => c.CustomerID == custId1);

// Pick some order

Order o = cust.Orders[0];

// Remove it directly from the table (I want it gone!)

db.Orders.Remove(o);

// Displays 'false'.. gone from customer's Orders

Console.WriteLine(cust.Orders.Contains(o));

// Displays 'true'.. order is detached from its customer

Console.WriteLine(o.Customer == null);

**Visual Basic**

Dim targetCustomer = (From cust In db.Customers \_

Where cust.CustomerID = custId1).First

' Pick some order

Dim o As Order = targetCustomer.Orders(0)

' Remove it directly from the table (I want it gone!)

db.Orders.Remove(o)

' Displays 'False'.. gone from customer’s Orders

Msgbox(targetCustomer.Orders.Contains(o))

' Displays 'True'.. order is detached from its customer

Msgbox(o.Customer = Nothing)

Like with all other changes, the order has not actually been deleted. It just looks that way to us since it has been removed and detached from the rest of our objects. When the order object was removed from the **Orders** table, it was marked for deletion by the change tracking service. The actually deletion from the database will occur when the changes are submitted on a call to **SubmitChanges()**. Note that the object itself is never deleted. The runtime manages the lifetime of object instances, so it sticks around as long as you are still holding a reference to it. However, after an object has been removed from its **Table** and changes submitted it is no longer tracked by the change tracking service.

The only other time an entity is left untracked is when it exists before the **DataContext** is aware of it. This happens whenever you create new objects in your code. You are free to use instances of entity classes in your application without ever retrieving them from a database. Change tacking and identity management only apply to those objects that the **DataContext** is aware of. Therefore neither service is enabled for newly created instances until you add them to the **DataContext**.

This can occur in one of two ways. You can call the **Add()** method on the related **Table** collection manually.

**C#**

Customer cust =

new Customer {

CustomerID = "ABCDE",

ContactName = "Frond Smooty",

CompanyTitle = "Eggbert's Eduware",

Phone = "888-925-6000"

};

// Add new customer to Customers table

db.Customers.Add(cust);

**Visual Basic**

Dim targetCustomer = New Customer With { \_

.CustomerID = “ABCDE”, \_

.ContactName = “Frond Smooty”, \_

.CompanyTitle = “Eggbert’s Eduware”, \_

.Phone = “888-925-6000”}

' Add new customer to Customers table

db.Customers.Add(cust)

Alternatively, you can attach a new instance to an object that the **DataContext** is already aware of.

**C#**

// Add an order to a customer's Orders

cust.Orders.Add(

new Order { OrderDate = DateTime.Now }

);

**Visual Basic**

' Add an order to a customer's Orders

targetCustomer.Orders.Add( \_

New Order With { .OrderDate = DateTime.Now } )

The **DataContext** will discover your new object instances even if they are attached to other new instances.

**C#**

// Add an order and details to a customer's Orders

Cust.Orders.Add(

new Order {

OrderDate = DateTime.Now,

OrderDetails = {

new OrderDetail {

Quantity = 1,

UnitPrice = 1.25M,

Product = someProduct

}

}

}

);

**Visual Basic**

' Add an order and details to a customer's Orders

targetCustomer.Orders.Add( \_

New Order With { \_

.OrderDate = DateTime.Now, \_

.OrderDetails = New OrderDetail With { \_

.Quantity = 1,

.UnitPrice = 1.25M,

.Product = someProduct

}

} )

Basically, the **DataContext** will recognize any entity in your object graph that is not currently tracked as a new instance, whether or not you called the **Add()** method.

**Using a read-only DataContext**

Many scenarios don't necessitate updating the entities retrieved from the database. Showing a table of Customers on a Web page is one obvious example. In all such cases, it is possible to improve performance by instructing the **DataContext** not to track the changes to the entities. This is achieved by specifying the **ObjectTracking** property on the **DataContext** to be false as in the following code:

**C#**

db.ObjectTracking = false;

var q = db.Customers.Where( c => c.City = "London");

foreach(Customer c in q)

Display(c);

**Visual Basic**

db.ObjectTracking = False

Dim londonCustomers = From cust In db.Customer \_

Where cust.City = "London"

For Each c in londonCustomers

Display(c)

Next

**Submitting Changes**

Regardless of how many changes you make to your objects, those changes were only made to in-memory replicas. Nothing has yet happened to the actual data in the database. Transmission of this information to the server will not happen until you explicitly request it by calling **SubmitChanges()** on the **DataContext**.

**C#**

Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// make changes here

db.SubmitChanges();

**Visual Basic**

Dim db As New Northwind("c:\northwind\northwnd.mdf")

' make changes here

db.SubmitChanges()

When you do call **SubmitChanges()**, the **DataContext** will attempt to translate all your changes into equivalent SQL commands, inserting, updating, or deleting rows in corresponding tables. These actions can be overridden by your own custom logic if you desire, however the order of submission is orchestrated by a service of the **DataContext** known as the *change processor*.

The first thing that happens when you call **SubmitChanges()** is that the set of known objects are examined to determine if new instances have been attached to them. These new instances are added to the set of tracked objects. Next, all objects with pending changes are ordered into a sequence of objects based on dependencies between them. Those objects whose changes depend on other objects are sequenced after their dependencies. Foreign key constraints and uniqueness constraints in the database play a big part in determining the correct ordering of changes. Then, just before any actual changes are transmitted, a transaction is started to encapsulate the series of individual commands unless one is already in scope. Finally, one by one the changes to the objects are translated into SQL commands and sent to the server.

At this point, any errors detected by the database will cause the submission process to abort and an exception will be raised. All changes to the database will be rolled back as if none of the submissions ever took place. The **DataContext** will still have a full recording of all changes so it is possible to attempt to rectify the problem and resubmit them by calling **SubmitChanges()** again.

**C#**

Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// make changes here

try {

db.SubmitChanges();

}

catch (Exception e) {

// make some adjustments

...

// try again

db.SubmitChanges();

}

**Visual Basic**

Dim db As New Northwind("c:\northwind\northwnd.mdf")

' make changes here

Try

db.SubmitChanges()

Catch e As Exception

' make some adjustments

...

' try again

db.SubmitChanges()

End Try

When the transaction around the submission completes successfully, the **DataContext** will accept the changes to the objects by simply forgetting the change tracking information.

**Simultaneous Changes**

There are a variety of reasons why a call to **SubmitChanges()** may fail. You may have created an object with an invalid primary key; one that's already in use, or with a value that violates some check constraint of the database. These kinds of checks are difficult to bake into business logic since they often require absolute knowledge of the entire database state. However, the most likely reason for failure is simply that someone else made changes to the objects before you.

Certainly, this would be impossible if you were locking each object in the database and using a fully serialized transaction. However, this style of programming (*pessimistic concurrency*) is rarely used since it is expensive and true clashes seldom occur. The most popular form of managing simultaneous changes is to employ a form of *optimistic concurrency*. In this model, no locks against the database rows are taken at all. That means any number of changes to the database could have occurred between the time you first retrieved your objects and the time you submitted your changes.

Therefore, unless you want to go with a policy that the last update wins, wiping over whatever else occurred before you, you probably want to be alerted to the fact that the underlying data was changed by someone else.

The **DataContext** has built-in support for optimistic concurrency by automatically detecting change conflicts. Individual updates only succeed if the database's current state matches the state you understood the data to be in when you first retrieved your objects. This happens on a per object basis, only alerting you to violations if they happen to objects you have made changes to.

You can control the degree to which the **DataContext** detects change conflicts when you define your entity classes. Each **Column** attribute has a property called **UpdateCheck** that can be assigned one of three values: **Always**, **Never**, and **WhenChanged**. If not set the default for a **Column** attribute is **Always**, meaning the data values represented by that member are always checked for conflicts, that is, unless there is an obvious tie-breaker like a version stamp. A **Column** attribute has an **IsVersion** property that allows you to specify whether the data value constitutes a version stamp maintained by the database. If a version exists, then the version is used alone to determine if a conflict has occurred.

When a change conflict does occur, an exception will be thrown just as if it were any other error. The transaction surrounding the submission will abort, yet the **DataContext** will remain the same, allowing you the opportunity to rectify the problem and try again.

**C#**

while (retries < maxRetries) {

Northwind db = new Northwind("c:\\northwind\\northwnd.mdf");

// fetch objects and make changes here

try {

db.SubmitChanges();

break;

}

catch (ChangeConflictException e) {

retries++;

}

}

**Visual Basic**

Do While retries < maxRetries

Dim db As New Northwind("c:\northwind\northwnd.mdf")

' fetch objects and make changes here

Try

db.SubmitChanges()

Exit Do

catch cce As ChangeConflictException

retries += 1

End Try

Loop

If you are making changes on a middle-tier or server, the easiest thing you can do to rectify a change conflict is to simply start over and try again, recreating the context and reapplying the changes. Additional options are described in the following section.

**Transactions**

A transaction is a service provided by databases or any other resource manager that can be used to guarantee that a series of individual actions occur automatically; meaning either they all succeed or they all don't. If they don't, then they are also all automatically undone before anything else is allowed to happen. If no transaction is already in scope, the **DataContext** will automatically start a database transaction to guard updates when you call **SubmitChanges()**.

You may choose to control the type of transaction used, its isolation level or what it actually encompasses by initiating it yourself. The transaction isolation that the **DataContext** will use is known as **ReadCommitted**.

**C#**

Product prod = db.Products.Single(p => p.ProductID == 15);

if (prod.UnitsInStock > 0)

prod.UnitsInStock--;

using(TransactionScope ts = new TransactionScope()) {

db.SubmitChanges();

ts.Complete();

}

**Visual Basic**

Dim product = (From prod In db.Products \_

Where prod.ProductID = 15).First

If product.UnitsInStock > 0) Then

product.UnitsInStock -= 1

End If

Using ts As TransactionScope = New TransactionScope())

db.SubmitChanges()

ts.Complete()

End Using

The example above initiates a fully serialized transaction by creating a new transaction scope object. All database commands executed within the scope of the transaction will be guarded by the transaction.

**C#**

Product prod = db.Products.Single(p => p.ProductId == 15);

if (prod.UnitsInStock > 0)

prod.UnitsInStock--;

using(TransactionScope ts = new TransactionScope()) {

db.ExecuteCommand("exec sp\_BeforeSubmit");

db.SubmitChanges();

ts.Complete();

}

**Visual Basic**

Dim product = (From prod In db.Products \_

Where prod.ProductID = 15).First

If product.UnitsInStock > 0) Then

product.UnitsInStock -= 1

End If

Using ts As TransactionScope = New TransactionScope())

db.ExecuteCommand(“exec sp\_BeforeSubmit”)

db.SubmitChanges()

ts.Complete()

End Using

This modified version of the same example uses the **ExecuteCommand()** method on the **DataContext** to execute a stored procedure in the database right before the changes are submitted. Regardless of what the stored procedure does to the database, we can be certain its actions are part of the same transaction.

If the transaction completes successfully, the **DataContext** throws out all the accumulated tracking information and treats the new states of the entities as unchanged. It does not, however, rollback the changes to your objects if the transaction fails. This allows you the maximum flexibility in dealing with problems during change submission.

It is also possible to use a local SQL transaction instead of the new **TransactionScope**. LINQ to SQL offers this capability to help you integrate LINQ to SQL features into pre-existing ADO.NET applications. However, if you go this route you will need to be responsible for much more.

**C#**

Product prod = q.Single(p => p.ProductId == 15);

if (prod.UnitsInStock > 0)

prod.UnitsInStock--;

db.Transaction = db.Connection.BeginTransaction();

try {

db.SubmitChanges();

db.Transaction.Commit();

}

catch {

db.Transaction.Rollback();

throw;

}

finally {

db.Transaction = null;

}

**Visual Basic**

Dim product = (From prod In db.Products \_

Where prod.ProductID = 15).First

If product.UnitsInStock > 0) Then

product.UnitsInStock -= 1

End If

db.Transaction = db.Connection.BeginTransaction()

Try

db.SubmitChanges()

db.Transaction.Commit()

catch e As Exception

db.Transaction.Rollback()

Throw e

Finally

db.Transaction = Nothing

End Try

As you can see, using a manually controlled database transaction is a bit more involved. Not only do you have to start it yourself, you have to tell the **DataContext** explicitly to use it by assigning it to the **Transaction** property. Then you must use a try-catch block to encase your submit logic, remembering to explicitly tell the transaction to commit and to explicitly tell the **DataContext** to accept changes, or to abort the transactions if there is failure at any point. Also, don't forget to set the **Transaction** property back to **null** when you are done.

**Stored Procedures**

When **SubmitChanges()** is called, LINQ to SQL generates and executes SQL commands to insert, update, and delete rows in the database. These actions can be overridden by application developers and in their place custom code can be used to perform the desired actions. In this way, alternative facilities like database-stored procedures can be invoked automatically by the *change processor*.

Consider a stored procedure for updating the units in stock for the Products table in the Northwind sample database. The SQL declaration of the procedure is as follows.

**SQL**

create proc UpdateProductStock

@id int,

@originalUnits int,

@decrement int

as

You can use the stored procedure instead of the normal auto-generated update command by defining a method on your strongly typed **DataContext**. Even if the **DataContext** class is being auto-generated by the LINQ to SQL code generation tool, you can still specify these methods in a partial class of your own.

**C#**

public partial class Northwind : DataContext

{

...

public void UpdateProduct(Product original, Product current) {

// Execute the stored procedure for UnitsInStock update

if (original.UnitsInStock != current.UnitsInStock) {

int rowCount = this.ExecuteCommand(

"exec UpdateProductStock " +

"@id={0}, @originalUnits={1}, @decrement={2}",

original.ProductID,

original.UnitsInStock,

(original.UnitsInStock - current.UnitsInStock)

);

if (rowCount < 1)

throw new Exception("Error updating");

}

...

}

}

**Visual Basic**

Partial Public Class Northwind

Inherits DataContext

...

Public Sub UpdateProduct(original As Product, current As Product)

‘ Execute the stored procedure for UnitsInStock update

If original.UnitsInStock <> current.UnitsInStock Then

Dim rowCount As Integer = ExecuteCommand( \_

"exec UpdateProductStock " & \_

"@id={0}, @originalUnits={1}, @decrement={2}", \_

original.ProductID, \_

original.UnitsInStock, \_

(original.UnitsInStock - current.UnitsInStock) )

If rowCount < 1 Then

Throw New Exception(“Error updating”)

End If

End If

...

End Sub

End Class

The signature of the method and the generic parameter tells the **DataContext** to uses this method in place of a generated update statement. The original and current parameters are used by LINQ to SQL for passing in the original and current copies of the object of the specified type. The two parameters are available for optimistic concurrency conflict detection.

**Note**   If you override the default update logic, conflict detection is your responsibility.

The stored procedure **UpdateProductStock** is invoked using the **ExecuteCommand()** method of the **DataContext**. It returns the number of rows affected and has the following signature:

**C#**

public int ExecuteCommand(string command, params object[] parameters);

**Visual Basic**

Public Function ExecuteCommand(command As String, \_

ParamArray parameters() As Object) As Integer

The object array is used for passing parameters required for executing the **command**.

Similar to the update method, insert and delete methods may be specified. Insert and delete methods take only one parameter of the entity type to be updated. For example, methods to insert and delete a Product instance can be specified as follows:

**C#**

public void InsertProduct(Product prod) { ... }

public void DeleteProudct(Product prod) { ... }

**Visual Basic**

Public Sub InsertProduct(prod As Product) ...

Public Sub DeleteProudct(prod As Product) ...

**Entity Classes In-Depth**

**Using Attributes**

An entity class is just like any normal object class that you might define as part of your application, except that it is annotated with special information that associates it with a particular database table. These annotations are made as custom attributes on your class declaration. The attributes are only meaningful when you use the class in conjunction with LINQ to SQL. They are similar to the XML serialization attributes in the .NET Framework. These "data" attributes provide LINQ to SQL with enough information to translate queries for your objects into SQL queries against the database and changes to your objects into SQL insert, update, and delete commands.

It is also possible to represent the mapping information by using an XML mapping file instead of attributes. This scenario is described in more detail in the [External Mapping](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_topic35) section.

**Database Attribute**

The **Database** attribute is used to specify the default name of database if it is not supplied by the connection. **Database** attributes can be applied to strongly typed **DataContext** declarations. This attribute is optional.

**Database Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | Specifies the name of the database. The information is used only if the connection itself does not specify the database name. If this Database attribute does not exist on context declaration and one is not specified by the connection, then database is assumed to have the same name as the context class. |

**C#**

[Database(Name="Database#5")]

public class Database5 : DataContext {

...

}

**Visual Basic**

<Database(Name:="Database#5")> \_

Public Class Database5

Inherits DataContext

...

End Class

**Table Attribute**

The **Table** attribute is used to designate a class as an entity class associated with a database table. Classes with the **Table** attribute will be treated specially by LINQ to SQL.

**Table Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | Specifies the name of the table. If this information is not specified it is assumed that the table has the same name as the entity class. |

**C#**

[Table(Name="Customers")]

public class Customer {

...

}

**Visual Basic**

<Table(Name:="Customers")> \_

Public Class Customer

...

End Class

**Column Attribute**

The **Column** attribute is used to designate a member of an entity class that represents a column in a database table. It can be applied to any field or property, public, private or internal. Only members identified as columns are persisted when LINQ to SQL saves changes to the database.

**Column Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | The name of the column in the table or view. If not specified the column is assumed to have the same name as the class member. |
| Storage | String | The name of the underlying storage. If specified it tells LINQ to SQL how to bypass the public property accessor for the data member and interact with the raw value itself. If not specified LINQ to SQL gets and sets the value using the public accessor. |
| DBType | String | The type of database column specified using database types and modifiers. This will be the exact text used to define the column in a T-SQL table declaration command. If not specified the database column type is inferred from the member type. The specific database type is only necessary if **CreateDatabase()** method is expected to be used to create an instance of the database. |
| IsPrimaryKey | Bool | If set to **true**, the class member represents a column that is part of the table's primary key. If more than one member of the class is designated as the Id, the primary key is said to be a composite of the associated columns. |
| IsDbGenerated | Boolean | Identifies that the member's column value is auto-generated by the database. Primary keys that are designated **IsDbGenerated=true** should also have a **DBType** with the **IDENTITY** modifier. **IsDbGenerated** members are synchronized immediately after the data row is inserted and are available after **SubmitChanges()** completes. |
| IsVersion | Boolean | Identifies the member's column type as a database timestamp or a version number. Version numbers are incremented and timestamp columns are updated by the database every time the associated row is updated. Members with **IsVersion=true** are synchronized immediately after the data row is updated. The new values are visible after **SubmitChanges()** completes. |
| UpdateCheck | UpdateCheck | Determines how LINQ to SQL implements *optimistic concurrency* conflict detection. If no member is designate as **IsVersion=true** detection is done by comparing original member values with current database state. You can control which members LINQ to SQL uses during conflict detection by giving each member an **UpdateCheck** enum value.   * **Always**: always use this column for conflict detection * **Never**: never use this column for conflict detection * **WhenChanged**: only use this column when the member has been changed by the application |
| IsDiscriminator | Boolean | Determines if the class member holds the discriminator value for an inheritance hierarchy. |
| Expression | String | Does not affect LINQ to SQL's operation, but is used during .**CreateDatabase()** as a raw SQL expression representing the computed column expression. |
| CanBeNull | Boolean | Indicates that the value can contain the null value. This is usually inferred from the CLR type of the entity member. Use this attribute to indicate that a string value is represented as a not nullable column in the database. |
| AutoSync | AutoSync | Specifies if the column is automatically synchronized from the value generated by the database on insert or update commands. Valid values for this tag are **OnInsert**, **Always**, and **Never**. |

A typical entity class will use **Column** attributes on public properties and store actual values in private fields.

**C#**

private string \_city;

[Column(Storage="\_city", DBType="NVarChar(15)")]

public string City {

get { ... }

set { ... }

}

**Visual Basic**

Private \_city As String

<Column(Storage:="\_city", DBType:="NVarChar(15)")> \_

public Property City As String

Get

set

End Property

The **DBType** is only specified so that the **CreateDatabase()** method can construct the table with the most precise type. Otherwise, the knowledge that the underlying column is limited to 15 characters is unused.

Members representing the primary key of a database type will often be associated with auto-generated values.

**C#**

private string \_orderId;

[Column(Storage="\_orderId", IsPrimaryKey=true, IsDbGenerated = true,

DBType="int NOT NULL IDENTITY")]

public string OrderId {

get { ... }

set { ... }

}

**Visual Basic**

Private \_orderId As String

<Column(Storage:="\_orderId", IsPrimaryKey:=true, \_

IsDbGenerated:= true, DBType:="int NOT NULL IDENTITY")> \_

public Property OrderId As String

Get

Set

End Property

If you do specify the **DBType**, make sure to include the **IDENTITY** modifier. LINQ to SQL will not augment a custom specified **DBType**. However, if the **DBType** is left unspecified LINQ to SQL will infer that the **IDENTITY** modifier is needed when creating the Database via the **CreateDatabase()** method.

Likewise, if the **IsVersion** property is true, the **DBType** must specify the correct modifiers to designate a version number or timestamp column. If no **DBType** is specified, LINQ to SQL will infer the correct modifiers.

You can control access to a member associated with an auto-generated column, version stamp, or any column you might want to hide by designating the access level of the member, or even limiting the accessor itself.

**C#**

private string \_customerId;

[Column(Storage="\_customerId", DBType="NCHAR(5) ")]

public string CustomerID {

get { ... }

}

**Visual Basic**

Private \_customerId As String

<Column(Storage:="\_customerId", DBType:="NCHAR(5)")> \_

Public Property CustomerID As String

Get

End Property

The Order's **CustomerID** property can be made read-only by not defining a set accessor. LINQ to SQL can still get and set the underlying value through the storage member.

You can also make a member completely inaccessible to the rest of the application by placing a Column attribute on a private member. This allows the entity class to contain information relevant to the class's business logic without exposing it in general. Even though private members are part of the translated data, since they are private you cannot refer to them in a language-integrated query.

By default, all members are used to perform optimistic concurrency conflict detection. You can control whether a particular member is used by specifying its **UpdateCheck** value.

**C#**

[Column(Storage="\_city", UpdateCheck=UpdateCheck.WhenChanged)]

public string City {

get { ... }

set { ... }

}

**Visual Basic**

<Column(Storage:="\_city", UpdateCheck:=UpdateCheck.WhenChanged)> \_

Public Property City As String

Get

Set

End Property

The following table shows the permissible mappings between database types and the corresponding CLR type. Use this table as a guide when determine which CLR type to use to represent a particular database column.

**Database Type and Corresponding CLR Type Permissible Mappings**

|  |  |  |
| --- | --- | --- |
| **Database Type** | **.NET CLR Type** | **Comments** |
| bit, tinyint, smallint, int, bigint | Bye, Int16, Uint16, Int32, Uint32, Int64, Uint64 | Lossy conversions possible. Values may not roundtrip. |
| bit | Boolean |  |
| decimal, numeric, smallmoney, money | Decimal | Scale difference may result in lossy conversion. May not roundtrip. |
| real, float | Single, Double | Precision differences. |
| char, varchar, text, nchar, nvarchar, ntext | String | Locale differences possible. |
| datetime, smalldatetime | DateTime | Different precision may cause lossy conversion and roundtrip problems. |
| uniqueidentifier | Guid | Different collation rules. Sorting may not work as expected. |
| timestamp | Byte[] (Byte() in Visual Basic), Binary | Byte array is treated as a scalar type. User is responsible for allocating adequate storage when constructor is called. It is considered immutable and is not tracked for changes. |
| binary, varbinary | Byte[] (Byte() in Visual Basic), Binary |  |

**Association Attribute**

The **Association** attribute is used to designate a property that represents a database association like a foreign-key to primary-key relationship.

**Association Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | The name of the association. This is often the same as the database's foreign-key constraint name. It is used when **CreateDatabase()** is used to create an instance of the database in order to generate the relevant constraint. It is also used to help distinguish between multiple relationships in a single entity class referring to the same target entity class. In this case, relationship properties on sides of the relationship (if both are defined) must have the same name. |
| Storage | String | The name of the underlying storage member. If specified it tells LINQ to SQL how to bypass the public property accessor for the data member and interact with the raw value itself. If not specified LINQ to SQL gets and sets the value using the public accessor. It is recommended that all association members be properties with separate storage members identified. |
| ThisKey | String | A comma-separated list of names of one or more members of this entity class that represent the key values on this side of the association. If not specified, the members are assumed to be the members that make up the primary key. |
| OtherKey | String | A comma-separated list of names of one or more members of the target entity class that represent the key values on the other side of the association. If not specified, the members are assumed to be the members that make up the other entity class's primary key. |
| IsUnique | Boolean | **True** if there a uniqueness constraint on the foreign key, indicating a true 1:1 relationship. This property is seldom used as 1:1 relationships are nearly impossible to manage within the database. Mostly entity models are defined using 1:n relationships even when they are treated as 1:1 by application developers. |
| IsForeignKey | Boolean | **True** if the target "other" type of the association is the parent of the source type. With foreign-key to primary-key relationships, the side holding the foreign-key is the child and the side holding the primary key is the parent. |
| DeleteRule | String | Used to add delete behavior to this association. For example, "CASCADE" would add "ON DELETE CASCADE" to the FK relationship. If set to **null**, no delete behavior is added. |

Association properties either represent a single reference to another entity class instance or they represent a collection of references. Singleton references must be encoded in the entity class using the **EntityRef<T>** (**EntityRef (OfT)** in Visual Basic) value type to store the actual reference. The **EntityRef** type is how LINQ to SQL enables *deferred loading* of references.

**C#**

class Order

{

...

private EntityRef<Customer> \_Customer;

[Association(Name="FK\_Orders\_Customers", Storage="\_Customer",

ThisKey="CustomerID")]

public Customer Customer {

get { return this.\_Customer.Entity; }

set { this.\_Customer.Entity = value;

// Additional code to manage changes }

}

}

**Visual Basic**

Class Order

...

Private \_customer As EntityRef(Of Customer)

<Association(Name:="FK\_Orders\_Customers", \_

Storage:="\_Customer", ThisKey:="CustomerID")> \_

public Property Customer() As Customer

Get

Return \_customer.Entity

End Get

Set (value As Customer)

\_customer.Entity = value

‘ Additional code to manage changes

End Set

End Class

The public property is typed as **Customer**, not **EntityRef<Customer>**. It is important not to expose the **EntityRef** type as part of the public API, as references to this type in a query will not be translated to SQL.

Likewise, an association property representing a collection must use the **EntitySet<T>** (**EntitySet(OfT)** in Visual Basic) collection type to store the relationship.

**C#**

class Customer

{

...

private EntitySet<Order> \_Orders;

[Association(Name="FK\_Orders\_Customers", Storage="\_Orders",

OtherKey="CustomerID")]

public EntitySet<Order> Orders {

get { return this.\_Orders; }

set { this.\_Orders.Assign(value); }

}

}

**Visual Basic**

Class Customer

...

Private \_Orders As EntitySet(Of Order)

<Association(Name:="FK\_Orders\_Customers", \_

Storage:="\_Orders", OtherKey:="CustomerID")> \_

public Property Orders() As EntitySet(Of Order)

Get

Return \_Orders

End Get

Set (value As EntitySet(Of Order))

\_Orders.Assign(value)

End Property

End Class

However, since an **EntitySet<T>** (**EntitySet(OfT)** in Visual Basic) is a collection, it is valid to use the **EntitySet** as the return type. It is also valid to disguise the true type of the collection, using the **ICollection<T>** (**ICollection(OfT)** in Visual Basic) interface instead.

**C#**

class Customer

{

...

private EntitySet<Order> \_Orders;

[Association(Name="FK\_Orders\_Customers", Storage="\_Orders",

OtherKey="CustomerID")]

public ICollection<Order> Orders {

get { return this.\_Orders; }

set { this.\_Orders.Assign(value); }

}

}

**Visual Basic**

Class Customer

...

Private \_orders As EntitySet(Of Order)

<Association(Name:="FK\_Orders\_Customers", \_

Storage:="\_Orders", OtherKey:="CustomerID")> \_

public Property Orders() As ICollection (Of Order)

Get

Return \_orders

End Get

Set (value As ICollection (Of Order))

\_orders.Assign(value)

End Property

End Class

Make certain to use the **Assign()** method on the **EntitySet** if you expose a public setter for the property. This allows the entity class to keep using the same collection instance since it may already be tied into the *change tracking service*.

**ResultType Attribute**

This attribute specifies an element type of an enumerable sequence that can be returned from a function that has been declared to return the **IMultipleResults** interface. This attribute can be specified more than once.

**ResultType Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Type | Type | Type of the returned results. |

**StoredProcedure Attribute**

The **StoredProcedure** attribute is used to declare that a call to a method defined on the **DataContext** or **Schema** type is translated as a call to a database stored procedure.

**StoredProcedure Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | The name of the stored procedure in the database. If not specified the stored procedure is assumed to have the same name as the method |

**Function Attribute**

The **Function** attribute is used to declare that a call to a method defined on a **DataContext** or **Schema** is translated as a call to a database user-defined scalar or table-valued function.

**Function Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | The name of the function in the database. If not specified the function is assumed to have the same name as the method |

**Parameter Attribute**

The **Parameter** attribute is used to declare a mapping between a method and the parameters of a database stored procedure or user-defined function.

**Parameter Attribute**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Name | String | The name of the parameter in the database. If not specified the parameter is inferred from the method parameter name. |
| DBType | String | The type of parameter specified using database types and modifiers. |

**InheritanceMapping Attribute**

The **InheritanceMapping** attribute is used to describe the correspondence between a particular discriminator codes and an inheritance subtype. All **InheritanceMapping** attributes used for an inheritance hierarchy must be declared on the root type of the hierarchy.

**InheritanceMapping Attribute**

|  |  |  |
| --- | --- | --- |
| **Propety** | **Type** | **Description** |
| Code | Object | The discriminator code value. |
| Type | Type | The Inheritance sub-type. This may be any non-abstract type in the inheritance hierarchy including the root type. |
| IsDefault | Boolean | Determines if the inheritance sub-type specified is the default type constructed when LINQ to SQL finds a discriminator code that is not defined by the **InheritanceMapping** attributes. Exactly one of the **InheritanceMapping** attributes must be declared with **IsDefault** as true. |

**Graph Consistency**

A graph is a general term for a data structure of objects all referring to each other by references. A hierarchy (or tree) is a degenerate form of graph. Domain-specific object models often describe a network of references that are best described as a graph of objects. The health of your object graph is vitally important to the stability of your application. That's why is important to make sure references within the graph remain consistent to your business rules and/or constraints defined in the database.

LINQ to SQL does not automatically manage consistency of relationship references for you. When relationships are bidirectional a change to one side of the relationship should automatically update the other. Note that it is uncommon for normal objects to behave this way so it is unlikely that you would have designed your objects this way otherwise.

LINQ to SQL does provide a few mechanisms to make this work easy and a pattern for you to follow to make sure you are managing your references correctly. Entity classes generated by the code generation tool will automatically implement the correct patterns.

**C#**

public class Customer() {

this.\_Orders =

new EntitySet<Order>(

new Action<Order>(this.attach\_Orders),

new Action<Order>(this.detach\_Orders));

);}

**Visual Basic**

Public Class Customer()

\_Orders = New EntitySet(Of Order)( \_

New Action(Of Order)(attach\_Orders), \_

New Action(Of Order)(detach\_Orders))

End Class

);}

The **EntitySet<T>** (**EntitySet(OfT)** in Visual Basic) type has a constructor that allows you to supply two delegates to be used as callbacks; the first when an item is added to the collection, the second when it is removed. As you can see from the example, the code you specify for these delegates can and should be written to update the reverse relationship property. This is how the **Customer** property on an **Order** instance is automatically changed when an order is added to a customer's **Orders** collection.

Implementing the relationship on the other end is not as easy. The **EntityRef<T>** (**EntityRef(OfT)** in Visual Basic) is a value type defined to contain as little additional overhead from the actual object reference as possible. It has no room for a pair of delegates. Instead, the code managing graph consistency of singleton references should be embedded in the property accessors themselves.

**C#**

[Association(Name="FK\_Orders\_Customers", Storage="\_Customer",

ThisKey="CustomerID")]

public Customer Customer {

get {

return this.\_Customer.Entity;

}

set {

Customer v = this.\_Customer.Entity;

if (v != value) {

if (v != null) {

this.\_Customer.Entity = null;

v.Orders.Remove(this);

}

this.\_Customer.Entity = value;

if (value != null) {

value.Orders.Add(this);

}

}

}

}

**Visual Basic**

<Association(Name:="FK\_Orders\_Customers", \_

Storage:="\_Customer", ThisKey:="CustomerID")> \_

Public Property Customer As Customer

Get

Return \_Customer.Entity

End Get

Set (value As Customer)

Dim cust As Customer v = \_customer.Entity

if cust IsNot value Then

If cust IsNot Nothing Then

\_Customer.Entity = Nothing

cust.Orders.Remove(Me)

End If

\_customer.Entity = value

if value IsNot Nothing Then

value.Orders.Add(Me)

End If

End If

End Set

End Property

Take a look at the setter. When the **Customer** property is being changed the order instance is first removed from the current customer's **Orders** collection and then only later added to the new customer's collection. Notice that before the call to **Remove()** is made the actual entity reference is set to **null**. This is done to avoid recursion when the **Remove()** method is called. Remember, the **EntitySet** will use callback delegates to assign this object's **Customer** property to **null**. The same thing happens right before the call to **Add()**. The actual entity reference is updated to the new value. This will again curtail any potential recursion and of course accomplish the task of the setter in the first place.

The definition of a one-to-one relationship is very similar to the definition of a one-to-many relationship from the side of the singleton reference. Instead of **Add()** and **Remove()** being called, a new object is assigned or a **null** is assigned to sever the relationship.

Again, it is vital that relationship properties maintain the consistency of the object graph. If the in-memory object graph is inconsistent with the database data, then a run-time exception is generated when the **SubmitChanges** method is called. Consider using the code generation tool to maintain consistency work for you.

**Change Notifications**

Your objects may participate in the change tracking process. It is not required that they do, but they can considerably reduce the amount of overhead needed to keep track of potential object changes. It is likely that your application will retrieve many more objects from queries than will end up being modified. Without proactive help from your objects, the change tracking service is limited in how it can actually track changes.

Since there is no true interception service in the runtime, the formal tracking does not actually occur. Instead, duplicate copies of the objects are stored when they are first retrieved. Later, when you call **SubmitChanges()**, these copies are used to compare against the ones you've been given. If their values differ, then the object has been modified. This means that every object requires two copies in memory even if you never change them.

A better solution is to have the objects themselves announce to the change tracking service when they are indeed changed. This can be accomplished by having the object implement an interface that exposes a callback event. The change tracking service can then wire up each object and receive notifications when they change.

**C#**

[Table(Name="Customers")]

public partial class Customer: INotifyPropertyChanging {

public event PropertyChangingEventHandler PropertyChanging;

private void OnPropertyChanging() {

if (this.PropertyChanging != null) {

this.PropertyChanging(this, emptyEventArgs);

}

}

private string \_CustomerID;

[Column(Storage="\_CustomerID", IsPrimaryKey=true)]

public string CustomerID {

get {

return this.\_CustomerID;

}

set {

if ((this.\_CustomerID != value)) {

this.OnPropertyChanging("CustomerID");

this.\_CustomerID = value;

}

}

}

}

**Visual Basic**

<Table(Name:="Customers")> \_

Partial Public Class Customer

Inherits INotifyPropertyChanging

Public Event PropertyChanging As PropertyChangingEventHandler \_

Implements INotifyPropertyChanging.PropertyChanging

Private Sub OnPropertyChanging()

RaiseEvent PropertyChanging(Me, emptyEventArgs)

End Sub

private \_customerID As String

<Column(Storage:="\_CustomerID", IsPrimaryKey:=True)>

public Property CustomerID() As String

Get

Return\_customerID

End Get

Set (value As Customer)

If \_customerID IsNot value Then

OnPropertyChanging(“CustomerID”)

\_CustomerID = value

End IF

End Set

End Function

End Class

To assist in improved change tracking, your entity classes must implement the **INotifyPropertyChanging** interface. It only requires you to define an event called **PropertyChanging**—the change tracking service then registers with your event when your objects come into its possession. All you are required to do is raise this event immediately before you are about to change a property's value.

Don't forget to put the same event raising logic in your relationship property setters, too. For **EntitySets**, raise the events in the delegates you supply.

**C#**

public Customer() {

this.\_Orders =

new EntitySet<Order>(

delegate(Order entity) {

this.OnPropertyChanging("Orders");

entity.Customer = this;

},

delegate(Order entity) {

this.onPropertyChanging("Orders");

entity.Customer = null;

}

);

}

**Visual Basic**

Dim \_orders As EntitySet(Of Order)

Public Sub New()

\_orders = New EntitySet(Of Order)( \_

AddressOf OrderAdding, AddressOf OrderRemoving)

End Sub

Sub OrderAdding(ByVal o As Order)

OnPropertyChanging()

o.Customer = Me

End Sub

Sub OrderRemoving(ByVal o As Order)

OnPropertyChanging()

o.Customer = Nothing

End Sub

**Inheritance**

LINQ to SQL supports single-table mapping, whereby an entire inheritance hierarchy is stored in a single database table. The table contains the flattened union of all possible data columns for the whole hierarchy and each row has nulls in the columns that are not applicable to the type of the instance represented by the row. The single-table mapping strategy is the simplest representation of inheritance and provides good performance characteristics for many different categories of queries.

**Mapping**

To implement this mapping using LINQ to SQL, you need to specify the following attributes and attribute properties on the root class of the inheritance hierarchy:

* The **[Table]** (**<Table>** in Visual Basic) attribute.
* An **[InheritanceMapping]** (**<InheritanceMapping>** in Visual Basic) attribute for each class in the hierarchy structure. For non-abstract classes, this attribute must define a Code property (a value that appears in the database table in the Inheritance Discriminator column to indicate which class or subclass this row of data belongs to) and a Type property (which specifies which class or subclass the key value signifies).
* An **IsDefault** property on a single **[InheritanceMapping]** (**<InheritanceMapping>** in Visual Basic) attribute. This property serves to designate a "fallback" mapping in case the discriminator value from the database table does not match any of the Code values in the inheritance mappings.
* An **IsDiscriminator** property for a **[Column]** (**<Column>** in Visual Basic) attribute, to signify that this is the column that holds the Code value for inheritance mapping.

No special attributes or properties are required on the subclasses. Note especially that subclasses do not have the **[Table]** (**<Table>** in Visual Basic) attribute.

In the following example, data contained in the **Car** and **Truck** subclasses are mapped to the single database table **Vehicle**. (To simplify the example, the sample code uses fields rather than properties for column mapping.)

**C#**

[Table]

[InheritanceMapping(Code = "C", Type = typeof(Car))]

[InheritanceMapping(Code = "T", Type = typeof(Truck))]

[InheritanceMapping(Code = "V", Type = typeof(Vehicle),

IsDefault = true)]

public class Vehicle

{

[Column(IsDiscriminator = true)]

public string Key;

[Column(IsPrimaryKey = true)]

public string VIN;

[Column]

public string MfgPlant;

}

public class Car : Vehicle

{

[Column]

public int TrimCode;

[Column]

public string ModelName;

}

public class Truck : Vehicle

{

[Column]

public int Tonnage;

[Column]

public int Axles;

}

**Visual Basic**

<Table> \_

<InheritanceMapping(Code:="C", Type:=Typeof(Car))> \_

<InheritanceMapping(Code:="T", Type:=Typeof(Truck))> \_

<InheritanceMapping(Code:="V", Type:=Typeof(Vehicle), \_

IsDefault:=true)> \_

Public Class Vehicle

<Column(IsDiscriminator:=True)> \_

Public Key As String

<Column(IsPrimaryKey:=True)> \_

Public VIN As String

<Column> \_

Public MfgPlant As String

End Class

Public Class Car

Inherits Vehicle

<Column> \_

Public TrimCode As Integer

<Column> \_

Public ModelName As String

End Class

Public class Truck

Inherits Vehicle

<Column> \_

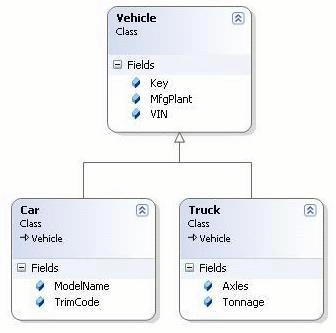
public Tonnage As Integer

<Column> \_

public Axles As Integer

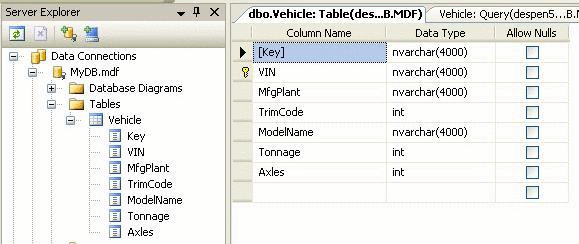
End Class

The class diagram appears as follows:



**Figure 1. Vehicle Class Diagram**

When you view the resulting database diagram in Server Explorer, you see that the columns have all been mapped to a single table, as shown here:



**Figure 2. Columns Mapped to a Single Table**

Note that the types of the columns that represent fields in the subtypes have to be nullable or they need to have a default specified. This is necessary for the insert commands to be successful.

**Querying**

The following code provides a flavor of how you can use derived types in your queries:

**C#**

var q = db.Vehicle.Where(p => p is Truck);

//or

var q = db.Vehicle.OfType<Truck>();

//or

var q = db.Vehicle.Select(p => p as Truck).Where(p => p != null);

foreach (Truck p in q)

Console.WriteLine(p.Axles);

**Visual Basic**

Dim trucks = From veh In db.Vehicle \_

Where TypeOf(veh) Is Truck

For Each truck In trucks

Console.WriteLine(p.Axles)

Next

**Advanced**

You can expand a hierarchy far beyond the simple sample already provided.

**Example 1**

Here is a much deeper hierarchy and more complex query:

**C#**

[Table]

[InheritanceMapping(Code = "V", Type = typeof(Vehicle), IsDefault = true)]

[InheritanceMapping(Code = "C", Type = typeof(Car))]

[InheritanceMapping(Code = "T", Type = typeof(Truck))]

[InheritanceMapping(Code = "S", Type = typeof(Semi))]

[InheritanceMapping(Code = "D", Type = typeof(DumpTruck))]

public class Truck: Vehicle { ... }

public class Semi: Truck { ... }

public class DumpTruck: Truck { ... }

...

// Get all trucks along with a flag indicating industrial application.

db.Vehicles.OfType<Truck>.Select(t =>

new {Truck=t, IsIndustrial=t is Semi || t is DumpTruck }

);

**Visual Basic**

<Table> \_

<InheritanceMapping(Code:="V", Type:=Typeof(Vehicle), IsDefault:=True)> \_

<InheritanceMapping(Code:="C", Type:=Typeof(Car))> \_

<InheritanceMapping(Code:="T", Type:=Typeof(Truck))> \_

<InheritanceMapping(Code:="S", Type:=Typeof(Semi))> \_

<InheritanceMapping(Code:="D", Type:=Typeof(DumpTruck))> \_

Public Class Truck

InheritsVehicle

Public Class Semi

Inherits Truck

Public Class DumpTruck

InheritsTruck

...

' Get all trucks along with a flag indicating industrial application.

Dim trucks = From veh In db.Vehicle \_

Where Typeof(veh) Is Truck And \_

IsIndustrial = (Typeof(veh) Is Semi \_

Or Typeof(veh) Is DumpTruck)

**Example 2**

The following hierarchy includes interfaces:

**C#**

[Table]

[InheritanceMapping(Code = "V", Type = typeof(Vehicle),

IsDefault = true)]

[InheritanceMapping(Code = "C", Type = typeof(Car))]

[InheritanceMapping(Code = "T", Type = typeof(Truck))]

[InheritanceMapping(Code = "S", Type = typeof(Semi))]

[InheritanceMapping(Code = "H", Type = typeof(Helicopter))]

public class Truck: Vehicle

public class Semi: Truck, IRentableVehicle

public class Helicopter: Vehicle, IRentableVehicle

**Visual Basic**

<Table> \_

<InheritanceMapping(Code:="V", Type:=TypeOf(Vehicle),

IsDefault:=True) > \_

<InheritanceMapping(Code:="C", Type:=TypeOf(Car)) > \_

<InheritanceMapping(Code:="T", Type:=TypeOf(Truck)) > \_

<InheritanceMapping(Code:="S", Type:=TypeOf(Semi)) > \_

<InheritanceMapping(Code:="H", Type:=TypeOf(Helicopter)) > \_

Public Class Truck

Inherits Vehicle

Public Class Semi

InheritsTruck, IRentableVehicle

Public Class Helicopter

InheritsVehicle, IRentableVehicle

Possible queries include the following:

**C#**

// Get commercial vehicles ordered by cost to rent.

db.Vehicles.OfType<IRentableVehicle>.OrderBy(cv => cv.RentalRate);

// Get all non-rentable vehicles

db.Vehicles.Where(v => !(v is IRentableVehicle));

**Visual Basic**

' Get commercial vehicles ordered by cost to rent.

Dim rentableVehicles = From veh In \_

db.Vehicles.OfType(Of IRentableVehicle).OrderBy( \_

Function(cv) cv.RentalRate)

' Get all non-rentable vehicles

Dim unrentableVehicles = From veh In \_

db.Vehicles.OfType(Of Vehicle).Where( \_

Function(uv) Not (TypeOf(uv) Is IRentableVehicle))

**Advanced Topics**

**Creating Databases**

Since entity classes have attributes describing the structure of the relational database tables and columns, it is possible to use this information to create new instances of your database. You can call the **CreateDatabase()** method on the **DataContext** to have LINQ to SQL construct a new database instance with a structure defined by your objects. There are many reasons you might want to do this: you might be building an application that automatically installs itself on a customer system, or a client application that needs a local database to save its offline state. For these scenarios, the **CreateDatabase()** is ideal—especially if a known data provider like SQL Server Express 2005 is available.

However, the data attributes may not encode everything about an existing database structure. The contents of user-defined functions, stored procedures, triggers, and check constraints are not represented by the attributes. The **CreateDatabase()** function will only create a replica of the database using the information it knows, which is the structure of the database and the types of columns in each table. Yet, for a variety of databases this is sufficient.

Below is an example of how you can create a new database named **MyDVDs.mdf**:

**C#**

[Table(Name="DVDTable")]

public class DVD

{

[Column(Id = true)]

public string Title;

[Column]

public string Rating;

}

public class MyDVDs : DataContext

{

public Table<DVD> DVDs;

public MyDVDs(string connection) : base(connection) {}

}

**Visual Basic**

<Table(Name:="DVDTable")> \_

Public Class DVD

<Column(Id:=True)> \_

public Title As String

<Column> \_

Public Rating As String

End Class

Public Class MyDVDs

Inherits DataContext

Public DVDs As Table(Of DVD)

Public Sub New(connection As String)

End Class

The object model can be used for creating a database using SQL Server Express 2005 as follows:

**C#**

MyDVDs db = new MyDVDs("c:\\mydvds.mdf");

db.CreateDatabase();

**Visual Basic**

Dim db As MyDVDs = new MyDVDs("c:\mydvds.mdf")

db.CreateDatabase()

LINQ to SQL also provides an API to drop an existing database prior to creating a new one. The database creation code above can be modified to first check for an existing version of the database using **DatabaseExists()** and then drop it using **DeleteDatabase()**.

**C#**

MyDVDs db = new MyDVDs("c:\\mydvds.mdf");

if (db.DatabaseExists()) {

Console.WriteLine("Deleting old database...");

db.DeleteDatabase();

}

db.CreateDatabase();

**Visual Basic**

Dim db As MyDVDs = New MyDVDs("c:\mydvds.mdf")

If (db.DatabaseExists()) Then

Console.WriteLine("Deleting old database...")

db.DeleteDatabase()

End If

db.CreateDatabase()

After the call to **CreateDatabase()**, the new database is able to accept queries and commands like **SubmitChanges()** to add objects to the MDF file.

It is also possible to use **CreateDatabase()** with a SKU other than SQL Server Express, using either an MDF file or just a catalog name. It all depends on what you use for your connection string. The information in the connection string is used to define the database that will exist, not necessarily one that already exists. LINQ to SQL will fish out the relevant bits of information and use it to determine what database to create and on what server to create it. Of course, you will need database admin rights or equivalent on the server to do so.

**Interoperating with ADO.NET**

LINQ to SQL is part of the ADO.NET family of technologies. It is based on services provided by the ADO.NET provider model, so it is possible to mix LINQ to SQL code with existing ADO.NET applications.

When you create a LINQ to SQL **DataContext**, you can supply it with an existing ADO.NET connection. All operations against the **DataContext**—including queries—will use the connection you provided. If the connection was already opened LINQ to SQL will honor your authority over the connection and leave it as is when finished with it. Normally LINQ to SQL closes its connection as soon as an operation is finished unless a transaction is in scope.

**C#**

SqlConnection con = new SqlConnection( ... );

con.Open();

...

// DataContext takes a connection

Northwind db = new Northwind(con);

...

var q =

from c in db.Customers

where c.City == "London"

select c;

**Visual Basic**

Dim con As SqlConnection = New SqlConnection( ... )

con.Open()

...

' DataContext takes a connection

Dim db As Northwind = new Northwind(con)

...

Dim q = From c In db.Customers \_

Where c.City = "London" \_

Select c

You can always access the connection used by your **DataContext** through the **Connection** property and close it yourself.

**C#**

db.Connection.Close();

**Visual Basic**

db.Connection.Close()

You can also supply the **DataContext** with your own database transaction, in case your application has already initiated one and you desire the **DataContext** to play along with it.

**C#**

IDbTransaction = con.BeginTransaction();

...

db.Transaction = myTransaction;

db.SubmitChanges();

db.Transaction = null;

**Visual Basic**

Dim db As IDbTransaction = con.BeginTransaction()

...

db.Transaction = myTransaction

db.SubmitChanges()

db.Transaction = Nothing

Whenever a **Transaction** is set, the **DataContext** will use it whenever it issues a query or executes a command. Don't forget to assign the property back to **null** when you are done.

However, the preferred method of doing transactions with the .NET Framework is to use the **TransactionScope** object. It allows you to make distributed transactions that work across databases and other memory resident resource managers. The idea is that transaction scopes start cheap, only promoting themselves to full on distributed transaction when they actually do refer to multiple databases or multiple connections within the scope of the transaction.

**C#**

using(TransactionScope ts = new TransactionScope()) {

db.SubmitChanges();

ts.Complete();

}

**Visual Basic**

Using ts As TransactionScope= New TransactionScope()

db.SubmitChanges()

ts.Complete()

End Using

**Executing SQL statements directly**

Connections and transactions are not the only way you can interoperate with ADO.NET. You might find that in some cases the query or submit changes facility of the **DataContext** is insufficient for the specialized task you may want to perform. In these circumstances it is possible to use the **DataContext** to issue raw SQL commands directly to the database.

The **ExecuteQuery()** method lets you execute a raw SQL query and converts the result of your query directly into objects. For example, assuming that the data for the **Customer** class is spread over two tables **customer1** and **customer2**, the following query returns a sequence of **Customer** objects.

**C#**

IEnumerable<Customer> results = db.ExecuteQuery<Customer>(

@"select c1.custid as CustomerID, c2.custName as ContactName

from customer1 as c1, customer2 as c2

where c1.custid = c2.custid"

);

**Visual Basic**

Dim results As IEnumerable(Of Customer) = \_

db.ExecuteQuery(Of Customer)( \_

"select c1.custid as CustomerID, " & \_

"c2.custName as ContactName " & \_

"from customer1 as c1, customer2 as c2 "& \_

"where c1.custid = c2.custid" )

As long as the column names in the tabular results match column properties of your entity class LINQ to SQL will materialize your objects out of any SQL query.

The **ExecuteQuery()** method also allows parameters. In the following code, a parameterized query is executed:

**C#**

IEnumerable<Customer> results = db.ExecuteQuery<Customer>(

"select contactname from customers where city = {0}",

"London"

);

**Visual Basic**

Dim results As IEnumerable(Of Customer) = \_

db.ExecuteQuery(Of Customer)( \_

"select contactname from customers where city = {0}", \_

"London" )

The parameters are expressed in the query text using the same curly notation used by **Console.WriteLine()** and **String.Format()**. In fact, **String.Format()** is actually called on the query string you provide, substituting the curly braced parameters with generated parameter names like **@p0**, **@p1 ...**, **@p(n)**.

**Change Conflict Resolution**

**Description**

A *change conflict* occurs when the client attempts to submit changes to an object and one or more values used in the update check have been updated in the database since the client last read them.

**Note**   Only members mapped as **UpdateCheck.Always** or **UpdateCheck.WhenChanged** participate in optimistic concurrency checks. No check is performed for members marked **UpdateCheck.Never**.

Resolution of this conflict includes discovering which members of the object are in conflict, and then deciding what to do about it. Note that optimistic concurrency might not be the best strategy in your particular situation. Sometimes it is perfectly reasonable to "let the last update win".

**Detecting, Reporting, and Resolving Conflicts in LINQ to SQL**

Conflict resolution is the process of refreshing a conflicting item by querying the database again and reconciling any differences. When an object is refreshed, the change tracker has the old original values and the new database values. LINQ to SQL then determines whether the object is in conflict or not. If it is, LINQ to SQL determines which members are involved. If the new database value for a member is different from the old original (which was used for the update check that failed), this is a conflict. Any member conflicts are added to a conflict list.

For example, in the following scenario, User1 begins to prepare an update by querying the database for a row. Before User1 can submit the changes, User2 has changed the database. User1's submission fails because the values expected for Col B and Col C have changed.

**Database Update Conflict**

|  |  |  |  |
| --- | --- | --- | --- |
| **User** | **Col A** | **Col B** | **Col C** |
| Original state | Alfreds | Maria | Sales |
| User 1 | Alfred |  | Marketing |
| User 2 |  | Mary | Service |

In LINQ to SQL, objects that fail to update because of optimistic concurrency conflicts cause an exception (**ChangeConflictException**) to be thrown. You can specify whether the exception should be thrown at the first failure or whether all updates should be attempted with any failures being accumulated and reported in the exception.

// [C#]

db.SubmitChanges(ConflictMode.FailOnFirstConflict);

db.SubmitChanges(ConflictMode.ContinueOnConflict);

' [Visual Basic]

db.SubmitChanges(ConflictMode.FailOnFirstConflict)

db.SubmitChanges(ConflictMode.ContinueOnConflict)

When thrown, the exception provides access to an **ObjectChangeConflict** collection. Details are available for each conflict (mapped to a single failed update attempt), including access to the **MemberConflicts** list. Each member conflict maps to a single member in the update that failed the concurrency check.

**Conflict Handling**

In the preceding scenario, User1 has the **RefreshMode** options described below for reconciling the differences before attempting to resubmit. In all cases, the record on the client is first "refreshed" by pulling down the updated data from the database. This action ensures that the next update attempt will not fail on the same concurrency checks.

Here, User1 chooses to merge database values with the current client values so that the database values are overwritten only when the current changeset has also modified that value. (See [Example 1](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_ex1) later in this section.)

In the scenario above, after conflict resolution, the result in the database is as follows:

**KeepChanges**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col A** | **Col B** | **Col C** |
| KeepChanges | Alfred (User 1) | Mary (User 2) | Marketing (User 1) |

* Col A: User1's change (Alfred) appears.
* Col B: User2's change (Mary) appears. This value was merged because User1 has not changed it.
* Col C: User1's change (Marketing) appears. User2's change (Service) is not merged because User1 has also changed that item.

Below, User1 chooses to overwrite any database values with the current values. (See [Example 2](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_ex2) later in this section.)

After the refresh, User1's changes are submitted. The result in the database is as follows:

**KeepCurrentValues**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col A** | **Col B** | **Col C** |
| KeepCurrentValues | Alfred (User 1) | Maria (Original) | Marketing (User 1) |

* Col A: User1's change (Alfred) appears.
* Col B: The original Maria remains; User2's change is discarded.
* Col C: User1's change (Marketing) appears. User2's change (Service) is discarded.

In the next scenario, User1 chooses to allow the database values to overwrite the current values in the client. (See [Example 3](http://msdn.microsoft.com/en-us/library/bb425822.aspx#linqtosql_ex3) later in this section.)

In the scenario above, after conflict resolution, the result in the database is as follows:

**OverwriteCurrentValues**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col A** | **Col B** | **Col C** |
| OverwriteCurrentValues | Alfreds (Original) | Mary (User 2) | Service (User 2) |

* Col A: The original value (Alfreds) remains; User1's value (Alfred) is discarded.
* Col B: User2's change (Mary) appears.
* Col C: User2's change (Service) appears. User1's change (Marketing) is discarded.

After conflicts have been resolved, you can attempt a resubmit. Because this second update might also fail, consider using a loop for update attempts.

**Examples**

The following code excerpts show various informational members and techniques at your disposal for discovering and resolving member conflicts.

**Example 1**

In this example, conflicts are resolved "automatically." That is, database values are merged with the current client values unless the client has also changed that value (**KeepChanges**). No inspection or custom handling of individual member conflicts takes place.

**C#**

try {

context.SubmitChanges(ConflictMode.ContinueOnConflict);

}

catch (ChangeConflictException e) {

//automerge database values into current for members

//that client has not modified

context.ChangeConflicts.Resolve(RefreshMode.KeepChanges);

}

//submit succeeds on second try

context.SubmitChanges(ConflictMode.FailOnFirstConflict);

**Visual Basic**

Try

context.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException

' automerge database values into current for members

' that client has not modified context.ChangeConflicts.Resolve(RefreshMode.KeepChanges)

End Try

' submit succeeds on second try

context.SubmitChanges(ConflictMode.FailOnFirstConflict)

**Example 2**

In this example, conflicts are resolved again without any custom handling. But this time, database values are not merged into current client values.

**C#**

try {

context.SubmitChanges(ConflictMode.ContinueOnConflict);

}

catch (ChangeConflictException e) {

foreach (ObjectChangeConflict cc in context.ChangeConflicts) {

//No database values are automerged into current

cc.Resolve(RefreshMode.KeepCurrentValues);

}

}

**Visual Basic**

Try

context.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException

For Each cc As ObjectChangeConflict In context.ChangeConflicts

‘No database values are automerged into current

cc.Resolve(RefreshMode.KeepCurrentValues)

Next

End Try

**Example 3**

Here again, no custom handling takes place. But in this case, all client values are updated with the current database values.

**C#**

try {

context.SubmitChanges(ConflictMode.ContinueOnConflict);

}

catch (ChangeConflictException e) {

foreach (ObjectChangeConflict cc in context.ChangeConflicts) {

//No database values are automerged into current

cc.Resolve(RefreshMode.OverwriteCurrentValues);

}

}

**Visual Basic**

Try

context.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException

For Each cc As ObjectChangeConflict In context.ChangeConflicts

' No database values are automerged into current

cc.Resolve(RefreshMode. OverwriteCurrentValues)

Next

End Try

**Example 4**

This example shows a way of accessing information on an entity in conflict.

**C#**

try {

user1.SubmitChanges(ConflictMode.ContinueOnConflict);

}

catch (ChangeConflictException e) {

Console.WriteLine("Optimistic concurrency error");

Console.ReadLine();

foreach (ObjectChangeConflict cc in user1.ChangeConflicts) {

ITable table = cc.Table;

Customers entityInConflict = (Customers)cc.Object;

Console.WriteLine("Table name: {0}", table.Name);

Console.Write("Customer ID: ");

Console.WriteLine(entityInConflict.CustomerID);

}

}

**Visual Basic**

Try

context.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException

Console.WriteLine("Optimistic concurrency error")

Console.ReadLine()

For Each cc As ObjectChangeConflict In context.ChangeConflicts

Dim table As ITable = cc.Table

Dim entityInConflict As Customers = CType(cc.Object, Customers)

Console.WriteLine("Table name: {0}", table.Name)

Console.Write("Customer ID: ")

Console.WriteLine(entityInConflict.CustomerID)

Next

End Try

**Example 5**

This example adds a loop through the individual members. Here you could provide custom handling of any member.

**Note**   Add **using System.Reflection**; to provide **MemberInfo**.

**C#**

try {

user1.SubmitChanges(ConflictMode.ContinueOnConflict);

}

catch (ChangeConflictException e) {

Console.WriteLine("Optimistic concurrency error");

Console.ReadLine();

foreach (ObjectChangeConflict cc in user1.ChangeConflicts) {

ITable table = cc.Table;

Customers entityInConflict = (Customers)cc.Object;

Console.WriteLine("Table name: {0}", table.Name);

Console.Write("Customer ID: ");

Console.WriteLine(entityInConflict.CustomerID);

foreach (MemberChangeConflict mc in cc.MemberConflicts) {

object currVal = mc.CurrentValue;

object origVal = mc.OriginalValue;

object databaseVal = mc.DatabaseValue;

MemberInfo mi = mc. Member;

Console.WriteLine("Member: {0}", mi.Name);

Console.WriteLine("current value: {0}", currVal);

Console.WriteLine("original value: {0}", origVal);

Console.WriteLine("database value: {0}", databaseVal);

Console.ReadLine();

}

}

}

**Visual Basic**

Try

user1.SubmitChanges(ConflictMode.ContinueOnConflict)

Catch e As ChangeConflictException

Console.WriteLine("Optimistic concurrency error")

Console.ReadLine()

For Each cc As ObjectChangeConflict In context.ChangeConflicts

Dim table As ITable = cc.Table

Dim entityInConflict As Customers = CType(cc.Object, Customers)

Console.WriteLine("Table name: {0}", table.Name)

Console.Write("Customer ID: ")

Console.WriteLine(entityInConflict.CustomerID)

For Each mc As MemberChangeConflict In cc.MemberConflicts

Dim currVal As Object = mc.CurrentValue

Dim origVal As Object = mc.OriginalValue

Dim databaseVal As Object = mc.DatabaseValue

Dim mi As MemberInfo = mc.Member

Console.WriteLine("Member: {0}", mi.Name)

Console.WriteLine("current value: {0}", currVal)

Console.WriteLine("original value: {0}", origVal)

Console.WriteLine("database value: {0}", databaseVal)

Console.ReadLine()

Next

Next

End Try

**Stored Procedures Invocation**

LINQ to SQL supports stored procedures and user-defined functions. LINQ to SQL maps these database-defined abstractions to code-generated client objects, so that you can access them in a strongly typed manner from client code. You can easily discover these methods using IntelliSense, and the method signatures resemble as closely as possible the signatures of the procedures and functions defined in the database. A result set returned by a call to a mapped procedure is a strongly typed collection. LINQ to SQL can automatically generate the mapped methods, but also supports manual mapping in situations where you choose not to use code generation.

LINQ to SQL maps stored procedures and functions to methods through the use of attributes. The **StoredProcedure**, **Parameter**, and **Function** attributes all support a *Name* property, and the **Parameter** attribute also supports a *DBType* property. Here are two examples:

**C#**

[StoredProcedure()]

public IEnumerable<CustOrderHistResult> CustOrderHist(

[Parameter(Name="CustomerID", DBType="NChar(5)")] string customerID) {

IExecuteResult result = this.ExecuteMethodCall(this,

((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID);

return ((IEnumerable<CustOrderHistResult>)(result.ReturnValue));

}

[Function(Name="[dbo].[ConvertTemp]")]

public string ConvertTemp(string string) { ... }

**Visual Basic**

<StoredProcedure()> \_

Public Function CustOrderHist( \_

<Parameter(Name:="CustomerID", DBType:="NChar(5)")> \_

customerID As String) As IEnumerable(Of CustOrderHistResult)

Dim result As IExecuteResult = ExecuteMethodCall(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), customerID)

Return CType(result.ReturnValue, IEnumerable(Of CustOrderHistResult))

End Function

<Function(Name:="[dbo].[ConvertTemp]")> \_

Public Function ConvertTemp(str As String) As String

The following examples show mappings for various kinds of stored procedures.

**Example 1**

The following stored procedure takes a single input parameter and returns an integer:

CREATE PROCEDURE GetCustomerOrderCount(@CustomerID nchar(5))

AS

Declare @count int

SELECT @count = COUNT(\*) FROM ORDERS WHERE CustomerID = @CustomerID

RETURN @count

The mapped method would be as follows:

**C#**

[StoredProcedure(Name = "GetCustomerOrderCount")]

public int GetCustomerOrderCount(

[Parameter(Name = "CustomerID")] string customerID) {

IExecuteResult result = this.ExecuteMethodCall(this,

((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID);

return (int) result.ReturnValue;

}

**Visual Basic**

<StoredProcedure (Name:="GetCustomerOrderCount")> \_

public Function GetCustomerOrderCount( \_

<Parameter(Name:= "CustomerID")> customerID As String) As Integer

Dim result As IExecuteResult = ExecuteMethodCall(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), customerID)

return CInt(result.ReturnValue)

End Function

**Example 2**

When a stored procedure can return multiple result shapes, the return type cannot be strongly typed to a single projection shape. In the following example, the result shape depends on the input:

CREATE PROCEDURE VariableResultShapes(@shape int)

AS

if(@shape = 1)

select CustomerID, ContactTitle, CompanyName from customers

else if(@shape = 2)

select OrderID, ShipName from orders

The mapped method is as follows:

**C#**

[StoredProcedure(Name = "VariableResultShapes")]

[ResultType(typeof(Customer))]

[ResultType(typeof(Order))]

public IMultipleResults VariableResultShapes(System.Nullable<int> shape) {

IExecuteResult result = this.ExecuteMethodCallWithMultipleResults(this,

((MethodInfo)(MethodInfo.GetCurrentMethod())), shape);

return (IMultipleResults) result.ReturnValue;

}

**Visual Basic**

<StoredProcedure(Name:= "VariableResultShapes")> \_

<ResultType(typeof(Customer))> \_

<ResultType(typeof(Order))> \_

public VariableResultShapes(shape As Integer?) As IMultipleResults

Dim result As IExecuteResult =

ExecuteMethodCallWithMultipleResults(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), shape)

return CType(result.ReturnValue, IMultipleResults)

End Function

You could use this stored procedure as follows:

**C#**

IMultipleResults result = db.VariableResultShapes(1);

foreach (Customer c in result.GetResult<Customer>()) {

Console.WriteLine(c.CompanyName);

}

result = db.VariableResultShapes(2);

foreach (Order o in result.GetResult<Order>()) {

Console.WriteLine(o.OrderID);

}

**Visual Basic**

Dim result As IMultipleResults = db.VariableResultShapes(1)

For Each c As Customer In result.GetResult(Of Customer)()

Console.WriteLine(c.CompanyName)

Next

result = db.VariableResultShapes(2);

For Each o As Order In result.GetResult(Of Order)()

Console.WriteLine(o.OrderID)

Next

}

Here you need to use the **GetResult** pattern to get an enumerator of the correct type, based on your knowledge of the stored procedure. LINQ to SQL can generate all possible projection types, but has no way of knowing in what order they will be returned. The only way you can know which generated projection types correspond to a mapped method is by using generated code comments on the methods.

**Example 3**

Here is the T-SQL of a stored procedure that returns multiple result shapes *sequentially*:

CREATE PROCEDURE MultipleResultTypesSequentially

AS

select \* from products

select \* from customers

LINQ to SQL would map this procedure just as in Example 2 above. In this case, however, there are two *sequential* resultsets.

**C#**

[StoredProcedure(Name="MultipleResultTypesSequentially")]

[ResultType(typeof(Product))]

[ResultType(typeof(Customer))]

public IMultipleResults MultipleResultTypesSequentially() {

return ((IMultipleResults)(

this.ExecuteMethodCallWithMultipleResults (this,

((MethodInfo)(MethodInfo.GetCurrentMethod()))).ReturnValue

)

);

}

**Visual Basic**

<StoredProcedure(Name:="MultipleResultTypesSequentially")> \_

<ResultType(typeof(Customer))> \_

<ResultType(typeof(Order))> \_

public Function MultipleResultTypesSequentially() As IMultipleResults

Return CType( ExecuteMethodCallWithMultipleResults (Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo)), \_

IMultipleResults).ReturnValue

End Function

You could use this stored procedure as follows:

**C#**

IMultipleResults sprocResults = db.MultipleResultTypesSequentially();

//first read products

foreach (Product p in sprocResults.GetResult<Product>()) {

Console.WriteLine(p.ProductID);

}

//next read customers

foreach (Customer c in sprocResults.GetResult<Customer>()){

Console.WriteLine(c.CustomerID);

}

**Visual Basic**

Dim sprocResults As IMultipleResults = db.MultipleResultTypesSequentially()

' first read products

For Each P As Product In sprocResults.GetResult(Of Product)()

Console.WriteLine(p.ProductID)

Next

' next read customers

For Each c As Customer c In sprocResults.GetResult(Of Customer)()

Console.WriteLine(c.CustomerID)

Next

**Example 4**

LINQ to SQL maps out parameters to reference parameters (**ref** keyword), and for value types declares the parameter as nullable (for example, **int?**). The procedure in the following example takes a single input parameter and returns an out parameter.

CREATE PROCEDURE GetCustomerCompanyName(

@customerID nchar(5),

@companyName nvarchar(40) output

)

AS

SELECT @companyName = CompanyName FROM Customers

WHERE CustomerID=@CustomerID

The mapped method is as follows:

**C#**

[StoredProcedure(Name = "GetCustomerCompanyName")]

public int GetCustomerCompanyName(

string customerID, ref string companyName) {

IExecuteResult result =

this.ExecuteMethodCall(this,

((MethodInfo)(MethodInfo.GetCurrentMethod())),

customerID, companyName);

companyName = (string)result.GetParameterValue(1);

return (int)result.ReturnValue;

}

**Visual Basic**

<StoredProcedure(Name:="GetCustomerCompanyName")> \_

Public Function GetCustomerCompanyName( \_

customerID As String, ByRef companyName As String) As Integer

Dim result As IExecuteResult = ExecuteMethodCall(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), customerID, \_

companyName)

companyName = CStr(result.GetParameterValue(1))

return CInt(result.ReturnValue)

End Function

In this case, the method does not have an explicit return value, but the default return value is mapped anyway. For the output parameter, a corresponding output parameter is used as expected.

You would call the above stored procedure as follows:

**C#**

string CompanyName = "";

string customerID = "ALFKI";

db.GetCustomerCompanyName(customerID, ref CompanyName);

Console.WriteLine(CompanyName);

**Visual Basic**

Dim CompanyName As String = ""

Dim customerID As String = "ALFKI"

db.GetCustomerCompanyName(customerID, CompanyName)

Console.WriteLine(CompanyName)

**User-defined Functions**

LINQ to SQL supports both scalar-valued and table-valued functions, and supports the in-line counterpart of both.

LINQ to SQL handles inline scalar calls similarly to the way system-defined functions are called. Consider the following query:

**C#**

var q =

from p in db.Products

select

new {

pid = p.ProductID,

unitp = Math.Floor(p.UnitPrice.Value)

};

**Visual Basic**

Dim productInfos = From prod In db.Products \_

Select p.ProductID, price = Math.Floor(p.UnitPrice.Value)

Here the method call **Math.Floor** is translated to a call to the system function **'FLOOR'**. In the same way, a call to a function that is mapped to a UDF is translated to a call to the UDF in SQL.

**Example 1**

Here is a scalar user-defined function (UDF) **ReverseCustName()**. In SQL Server, the function might be defined as follows:

CREATE FUNCTION ReverseCustName(@string varchar(100))

RETURNS varchar(100)

AS

BEGIN

DECLARE @custName varchar(100)

-- Impl. left as exercise for the reader

RETURN @custName

END

You can map a client method defined on a schema class to this UDF using the code below. Note that the body of the method constructs an expression that captures the intent of the method call, and passes that expression to the **DataContext** for translation and execution. (This direct execution happens only if the function is called.)

**C#**

[Function(Name = "[dbo].[ReverseCustName]")]

public string ReverseCustName(string string1) {

IExecuteResult result = this.ExecuteMethodCall(this,

(MethodInfo)(MethodInfo.GetCurrentMethod())), string1);

return (string) result.ReturnValue;

}

**Visual Basic**

Function(Name:= "[dbo].[ReverseCustName]")> \_

Public Function ReverseCustName(string1 As String) As String

Dim result As IExecuteResult = ExecuteMethodCall(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), string1)

return CStr(result.ReturnValue)

**Example 2**

In the following query, you can see an inline call to the generated UDF method **ReverseCustName**. In this case the function is not executed immediately. The SQL built for this query translates to a call to the UDF defined in the database (see the SQL code following the query).

**C#**

var q =

from c in db.Customers

select

new {

c.ContactName,

Title = db.ReverseCustName(c.ContactTitle)

};

**Visual Basic**

Dim customerInfos = From cust In db.Customers \_

Select c.ContactName, \_

Title = db.ReverseCustName(c.ContactTitle)

SELECT [t0].[ContactName],

dbo.ReverseCustName([t0].[ContactTitle]) AS [Title]

FROM [Customers] AS [t0]

When you call the same function *outside* a query, LINQ to SQL creates a simple query from the method call expression with the following SQL syntax (where the parameter @p0 is bound to the constant passed in):

In LINQ to SQL:

**C#**

string str = db.ReverseCustName("LINQ to SQL");

**Visual Basic**

Dim str As String = db.ReverseCustName("LINQ to SQL")

Converts to:

SELECT dbo.ReverseCustName(@p0)

**Example 3**

A table-valued function (TVF) returns a single result set (unlike stored procedures, which can return multiple result shapes). Because the TVF return type is table, you can use a TVF anywhere in SQL that you can use a table, and you can treat the TVF in the same way as you would a table.

Consider the following SQL Server definition of a table-valued function:

CREATE FUNCTION ProductsCostingMoreThan(@cost money)

RETURNS TABLE

AS

RETURN

SELECT ProductID, UnitPrice

FROM Products

WHERE UnitPrice > @cost

This function explicitly states that it returns a TABLE, so the returned result set structure is implicitly defined. LINQ to SQL maps the function as follows:

**C#**

[Function(Name = "[dbo].[ProductsCostingMoreThan]")]

public IQueryable<Product> ProductsCostingMoreThan(

System.Nullable<decimal> cost) {

return this.CreateMethodCallQuery<Product>(this,

(MethodInfo)MethodInfo.GetCurrentMethod(),

cost);

}

**Visual Basic**

<Function(Name:="[dbo].[ProductsCostingMoreThan]")> \_

Public Function ProductsCostingMoreThan(

cost As System.Nullable(Of Decimal)) As IQueryable(Of Product)

Return CreateMethodCallQuery(Of Product)(Me, \_

CType(MethodInfo.GetCurrentMethod(), MethodInfo), cost)

The following SQL code shows that you can join to the table returned by the function and otherwise treat it as you would any other table:

SELECT p2.ProductName, p1.UnitPrice

FROM dbo.ProductsCostingMoreThan(80.50)

AS p1 INNER JOIN Products AS p2 ON p1.ProductID = p2.ProductID

In LINQ to SQL, the query would be rendered as follows (using the new 'join' syntax):

**C#**

var q =

from p in db.ProductsCostingMoreThan(80.50m)

join s in db.Products on p.ProductID equals s.ProductID

select new {p.ProductID, s.UnitPrice};

**Visual Basic**

Dim productInfos = From costlyProd In db.ProductsCostingMoreThan(80.50m) \_

Join prod In db.Products \_

On costlyProd.ProductID Equals prod.ProductID \_

Select costlyProd.ProductID, prod.UnitPrice

**LINQ to SQL Limitations on Stored Procedures**

LINQ to SQL supports code generation for stored procedures that return statically determined result sets. Thus the LINQ to SQL code generator does not support the following:

* Stored procedures that use dynamic SQL to return result sets. When a stored procedure contains conditional logic to build a dynamic SQL statement, LINQ to SQL cannot acquire metadata for the resultset because the query used to generate the resultset is unknown until run time.
* Stored procedures that produce results based on temporary table.

**The Entity Class Generator Tool**

If you have an existing database, it is unnecessary to create a complete object model by hand just to represent it. The LINQ to SQL distribution comes with a tool called SQLMetal. It is a command-line utility that automates the task of creating entity classes by inferring the appropriate classes from the database metadata.

You can use SQLMetal to extract SQL metadata from a database and generate a source file containing entity class declarations. Alternatively, you can split the process into two steps, first generating an XML file representing the SQL metadata and then later translating that XML file into a source file containing class declarations. This split process allows you to retain the metadata as a file so you may edit it. The extraction process producing the file make a few inferences along the way about appropriate class and property names given the table and column names of the database. You might find it necessary to edit the XML file in order for the generator to produce more pleasing results or to hide aspects of the database that you don't want present in your objects.

The simplest scenario to use SQLMetal is to directly generate classes from an existing database. Here is how to invoke the tool:

**C#**

SqlMetal /server:.\SQLExpress /database:Northwind /pluralize /namespace:nwind /code:Northwind.cs

**Visual Basic**

SqlMetal /server:.\SQLExpress /database:Northwind /pluralize /namespace:nwind /code:Northwind.vb /language:vb

Executing the tool creates a **Northwind** **.cs** or **.vb** file that contains the object model generated by reading the database metadata. This usage works well if the names of the tables in the database are similar to the names of the objects that you want to generate. If not you'll want to take the two-step approach.

To instruct SQLMetal to generate a DBML file use the tool as follows:

SqlMetal /server:.\SQLExpress /database:Northwind /pluralize

/xml:Northwind.dbml

Once the dbml file is generated, you can go ahead and annotate it with **class** and **property** attribute to describe how tables and columns map to classes and properties. Once you have finished annotating the dbml file, you can generate your object model by running the following command:

**C#**

SqlMetal /namespace:nwind /code:Northwind.cs Northwind.dbml

**Visual Basic**

SqlMetal /namespace:nwind /code:Northwind.vb Northwind.dbml /language:vb

The SQLMetal usage signature is as follows:

SqlMetal [options] [filename]

The following is a table showing the available command line options for SQLMetal.

**Command Line Options for SQLMetal**

|  |  |
| --- | --- |
| **Option** | **Description** |
| /server:<*name*> | Indicates the server to connect to in order to access the database. |
| /database:<*name>* | Indicates the name of the database to read metadata from. |
| /user:<*name>* | Login user id for the server. |
| /password:<*name*> | Login password for the server. |
| /views | Extract database views. |
| /functions | Extract database functions. |
| /sprocs | Extract stored procedures. |
| /code[:<*filename*>] | Indicates that the output of the tool is a source file of entity class declarations. |
| /language:*<language>* | Use Visual Basic or C# (default). |
| /xml[:<*filename*>] | Indicates that the output of the tools is an DBML file describing the database metadata and the first guess approximation of class and property names. |
| /map[:<*filename*>] | Indicates that an external mapping file should be used instead of attributes. |
| /pluralize | Indicates that the tool should perform English language pluralizing / de-pluralizing heuristic to the names of the tables in order to produce appropriate class and property names. |
| /namespace:<*name*> | Indicates the namespace the entity classes will be generated in. |
| /timeout:<seconds> | Timeout value in seconds to use for database commands. |

**Note**   In order to extract the metadata from an MDF file, you must specify the MDF file name after all other options. If no **/server** is specified **localhost** is assumed.

**Generator Tool DBML Reference**

The DBML (Database Mapping Language) file is foremost a description of the SQL metadata for a given database. It is extracted by SQLMetal by looking at the database metadata. The same file is also used by SQLMetal to generate a default object model to represent the database.

Here is a prototypical example of the DBML syntax:

<?xml version="1.0" encoding="utf-16"?>

<Database Name="Northwind" EntityNamespace="Mappings.FunctionMapping"

ContextNamespace="Mappings.FunctionMapping"

Provider="System.Data.Linq.SqlClient.Sql2005Provider"

xmlns="http://schemas.microsoft.com/dsltools/LINQ to SQLML">

<Table Name="Categories">

<Type Name="Category">

<Column Name="CategoryID" Type="System.Int32"

DbType="Int NOT NULL IDENTITY" IsReadOnly="False"

IsPrimaryKey="True" IsDbGenerated="True" CanBeNull="False" />

<Column Name="CategoryName" Type="System.String"

DbType="NVarChar(15) NOT NULL" CanBeNull="False" />

<Column Name="Description" Type="System.String"

DbType="NText" CanBeNull="True" UpdateCheck="Never" />

<Column Name="Picture" Type="System.Byte[]"

DbType="Image" CanBeNull="True" UpdateCheck="Never" />

<Association Name="FK\_Products\_Categories" Member="Products"

ThisKey="CategoryID" OtherKey="CategoryID"

OtherTable="Products" DeleteRule="NO ACTION" />

</Type>

</Table>

<Function Name="GetCustomerOrders">

<Parameter Name="customerID" Type="System.String" DbType="NChar(5)" />

<ElementType Name="GetCustomerOrdersResult">

<Column Name="OrderID" Type="System.Int32"

DbType="Int" CanBeNull="True" />

<Column Name="ShipName" Type="System.String"

DbType="NVarChar(40)" CanBeNull="True" />

<Column Name="OrderDate" Type="System.DateTime"

DbType="DateTime" CanBeNull="True" />

<Column Name="Freight" Type="System.Decimal"

DbType="Money" CanBeNull="True" />

</ElementType>

</Function>

</Database>

The elements and their attributes are described as follows.

**Database**

This is the outermost element in the XML format. This element maps loosely to the Database attribute on the generated **DataContext**.

**Database Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | *None* | The name of the database. If present, and if generating a **DataContext**, will attach a Database attribute to it with this name. Also used as name of **DataContext** class if the class attribute is not present. |
| EntityNamespace | Strong | *None* | Default namespace for classes generated from Type elements within Table elements. If no namespace is specified here, entity classes are generated in the root namespace. |
| ContextNamespace | String | *None* | Default namespace for the generated **DataContext** class. If no namespace is specified here, that the **DataContext** class is generated in the root namespace. |
| Class | String | Database.Name | The name of the generated **DataContext** class. If no present, use the **Name** attribute of the Database element. |
| AccessModifier | AccessModifier | Public | The accessibility level of the generated **DataContext** class. Valid values are **Public**, **Protected**, **Internal** and **Private**. |
| BaseType | String | “System.Data.Linq.DataContext” | The base type of the **DataContext** class. |
| Provider | String | “System.Data.Linq.SqlClient.Sql2005Provider” | The provider of the **DataContext**, use Sql2005 provider as default |
| ExternalMapping | Boolean | False | Specify if the DBML is used for generating external mapping file. |
| Serialization | SerializationMode | SerializationMode.None | Specify if the generated **DataContext** and entity classes are serializable. |

**Database Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Element** | **Element Type** | **Occurrence Range** | **Description** |
| <Table> | Table | 0-unbounded | Represents a SQL Server table or view that will be mapped either to a single type or to an inheritance hierarchy. |
| <Function> | Function | 0-unbounded | Represents a SQL Server stored procedure or a db function that will be mapped to a method in the generated **DataContext** class. |
| <Connection> | Connection | 0-1 | Represents the database connection this **DataContext** will use. |

**Table**

This element represents a database table (or a view) that will be mapped either to a single type or to an inheritance hierarchy. This element maps loosely to the Table attribute on the generated entity class.

**Table Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | (*required*) | The name of the table within the database. Serves as the base of the default name for the table adapter, if needed. |
| Member | String | Table.Name | The name of the member field generated for this table within the **DataContext** class. |
| AccessModifier | AccessModifier | Public | The accessibility level of the **Table<T>** reference within the **DataContext**. Valid values are **Public**, **Protected**, **Internal** and **Private**. |

**Table Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Element** | **Element Type** | **Occurrence Range** | **Description** |
| <Type> | Type | 1-1 | Represents the type or inheritance hierarchy mapped to this table. |
| <InsertFunction> | TableFunction | 0-1 | The method for inserting. When it is present, a method **InsertT** is generated. |
| <UpdateFunction> | TableFunction | 0-1 | The method for updating. When it is present, a method **UpdateT** is generated. |
| <DeleteFunction> | TableFunction | 0-1 | The method for deleting. When it is present, a method **DeleteT** is generated. |

**Type**

This element represents a type definition for either a Table or a stored procedure result shape. This will code-gen into a new CLR type with the columns and associations specified.

Type may also represent a component of an inheritance hierarchy, with multiple types mapping to the same table. In this case the Type elements are nested to represent the parent-child inheritance relationships and are differentiated in the database by the **InheritanceCode** specified.

**Type Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | (required) | The name of the CLR type to be generated. |
| InheritanceCode | String | None | If this type is participating in inheritance, it can have an associated inheritance code to distinguish between CLR types when loading rows from the table. The **Type** whose **InheritanceCode** matches the value of the **IsDiscriminator** column is used to instantiate the loaded object. If the inheritance code is not present, the generated entity class is abstract. |
| IsInheritanceDefault | Boolean | False | If this is true for a **Type** in an inheritance hierarchy, this type will be used when loading rows that do not match on any defined inheritance codes. |
| AccessModifier | AccessModifier | Public | The accessibility level of the CLR type being created. Valid values are: **Public**, **Protected**, **Internal** and **Private**. |
| Id | String | None | A type can have a unique Id. Id of a type can be used by other tables or functions. Id only appears in DBML file, not in the object model. |
| IdRef | String | None | **IdRef** is used to refer to another type's **Id**. If **IdRef** is present in a type element, the type element must only contain the **IdRef** information. **IdRef** only appears in DBML file, not in the object model. |

**Type Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Element** | **Element Type** | **Occurrence Range** | **Description** |
| <Column> | Column | 0-unbounded | Represents a property within this type that will be bound to a field in this type's table. |
| <Association> | Association | 0-unbounded | Represents a property within this type that will be bound to one end of a foreign key relationship between tables. |
| <Type> | SubType | 0-unbounded | Represents subtypes of this type within an inheritance hierarchy. |

**SubType**

This element represents a derived type in an inheritance hierarchy. This will be generated into a new CLR type with the columns and associations specified in this type. No inheritance attributes are generated for subtypes.

Comparing to **Type**, **SubType** elements do not have **AccessModifier** because all derived types must be public. **SubTypes** cannot be reused by other tables and functions so there is no **Id** and **IdRef** in them.

**SubType Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | (*required*) | The name of the CLR type to be generated. |
| InheritanceCode | String | *None* | If this type is participating in inheritance, it can have an associated inheritance code to distinguish between CLR types when loading rows from the table. The Type whose **InheritanceCode** matches the value of the **IsDiscriminator** column is used to instantiate the loaded object. If the inheritance code is not present, the generated entity class is abstract. |
| IsInheritanceDefault | Boolean | False | If this is true for a **Type** in an inheritance hierarchy, this type will be used when loading rows that do not match on any defined inheritance codes. |

**SubType Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Element** | **Element Type** | **Occurrence Range** | **Description** |
| <Column> | Column | 0-unbounded | Represents a property within this type that will be bound to a field in this type's table. |
| <Association> | Association | 0-unbounded | Represents a property within this type that will be bound to on one end of a foreign key relationship between tables. |
| <Type> | SubType | 0-unbounded | Represents subtypes of this type within an inheritance hierarchy. |

**Column**

This element represents a column within a table that is mapped to a property (and backing field) within a class. There will be no **Column** element present for either end of a foreign key relationship, however, as that is completely represented (on both ends) by Association elements.

**Column Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attributes** | **Type** | **Default** | **Description** |
| Name | String | *None* | The name of the database field this column will map to. |
| Member | String | *Name* | The name of the CLR property to be generated on the containing type. |
| Storage | String | *\_Member* | The name of the private CLR backing field that will store this column's value. Do not remove **Storage** when serializing, even if it is default. |
| AccessModifier | AccessModifier | Public | The accessibility level of the CLR property being created. Valid values are: **Public**, **Protected**, **Internal** and **Private**. |
| Type | String | *(required)* | The name of the type of both the CLR property and backing field being created. This may be anything from a fully qualified name to just the direct name of a class, as long as the name will ultimately be in scope when the generated code is compiled. |
| DbType | String | *None* | Full SQL Server type (including annotation such as **NOT NULL**) for this column. Used by LINQ to SQL if you provide it to optimize the queries generated and to be more specific when doing **CreateDatabase()**. Always serialize **DbType**. |
| IsReadOnly | Boolean | False | If **IsReadOnly** is set, a property setter is not created, meaning people cannot change this column's value using that object. |
| IsPrimaryKey | Boolean | False | Indicates that this column participates in the primary key of the table. This information is required for LINQ to SQL to operate properly. |
| IsDbGenerated | Boolean | False | Indicates that this field's data is generated by the database. This is the case primarily for **AutoNumber** fields, and for calculated fields. It is not meaningful to assign values to these fields, and therefore they are automatically **IsReadOnly**. |
| CanBeNull | Boolean | *None* | Indicates that the value can contain the null value. If you wish to actually use null values in the CLR, you must still specify the **ClrType** as **Nullable<T>**. |
| UpdateCheck | UpdateCheck | Always (unless at least one other member has **IsVersion** set, then Never) | Indicates whether LINQ to SQL should use this column during optimistic concurrency conflict detection. Normally all columns participate by default, unless there is an **IsVersion** column, which then participates by itself. Can be: **Always**, **Never**, or **WhenChanged** (which means the column participates if its own value has changed). |
| IsDiscriminator | Boolean | False | Indicates whether this field contains the discriminator code used for choosing between types in an inheritance hierarchy. |
| Expression | String | *None* | Does not affect LINQ to SQL's operation, but is used during .**CreateDatabase()** as a raw SQL expression representing the computed column expression. |
| IsVersion | Boolean | False | Indicates that this field represents a **TIMESTAMP** field in SQL Server that is automatically updated each time the row is changed. This field can then be used to enable more efficient optimistic concurrency conflict detection. |
| IsDelayLoaded | Boolean | False | Indicates that this column should not be loaded immediately upon object materialization, but only when the relevant property is first accessed. This is useful for large memo fields or binary data in a row that is not always needed. |
| AutoSync | AutoSync | If (IsDbGenerated && IsPrimaryKey) OnInsert;  Else if (IsDbGenerated) Always  Else Never | Specifies if the column is automatically synchronized from the value generated by the database. Valid values for this tag are: **OnInsert**, **Always**, and **Never**. |

**Association**

This element represents either end of a foreign-key relationship. For one-to-many relationships, this will be an **EntitySet<T>** on the one side and an **EntityRef<T>** on the many side. For one-to-one relationships, this will be an **EntityRef<T>** on both sides.

Note that it is not required to have an **Association** entry on both sides of an association. In this case, a property will only be generated on the side that has the entry (forming a unidirectional relationship).

**Association Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | *(required)* | The name of the relation (usually the foreign key constraint name). This can technically be optional, but should always be generated by code to avoid ambiguity when there are multiple relationships between the same two tables. |
| Member | String | *Name* | The name of the CLR property to be generated on this side of the association. |
| Storage | String | *If OneToMany and Not IsForeignKey:*  *\_OtherTable*  *Else:*  *\_TypeName(OtherTable)* | The name of the private CLR backing field that will store this column's value. |
| AccessModifier | AccessModifier | Public | The accessibility level of the CLR property being created. Valid values are **Public**, **Protected**, **Internal** and **Private**. |
| ThisKey | String | *The IsIdentity property within the containing class* | A comma separated list of the keys on this side of the association. |
| OtherTable | String | *See description.* | The table on the other end of the relationship. Normally this can be determined by the LINQ to SQL runtime by matching relationship names, but this is not possible for unidirectional associations, or anonymous associations. |
| OtherKey | String | *The primary keys within the foreign class* | A comma separated list of the keys on the other side of the association. |
| IsForeignKey | Boolean | False | Indicates if this is the "child" side of the relationship, the many side of a one-to-many. |
| RelationshipType | RelationshipType | OneToMany | Indicates whether the user is asserting that the data related to by this association meets the criteria of one-to-one data or fits the more general case of one-to-many. For one-to-one, the user is asserting that for every row on the primary-key ("one") side, there is only one row on the foreign-key ("many") side. This will cause an **EntityRef<T>** to be generated on the "one" side instead of an **EntitySet<T>**. The valid values are **OneToOne** and **OneToMany**. |
| DeleteRule | String | *None* | Used to add delete behavior to this association. For example, "**CASCADE**" would add "**ON** **DELETE** **CASCADE**" to the FK relationship. If set to null, no delete behavior is added. |

**Function**

This element represents a stored procedure or a database function. For every **Function** node, a method is generated in the **DataContext** class.

**Function Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | *(required)* | The name of the stored procedure within the database. |
| Method | String | *Method* | The name of the CLR method to generate that allows invocation of the stored procedure. The default name for **Method** has things such as **[dbo].** stripped off **Name**. |
| AccessModifier | AccessModifier | Public | The accessibility level of the stored procedure method. Valid values are **Public**, **Protected**, **Internal** and **Private**. |
| HasMultipleResults | Boolean | # of Types > 1 | Specifies if the stored procedure represented by this **Function** node returns multiple resultsets. Every resultset is a tabular shape, it can either be an existing **Type** or be a set of columns. In the latter case, a **Type** node will be created for the column set. |
| IsComposable | Boolean | False | Specifies if the function/stored procedure can be composed in LINQ to SQL queries. Only DB functions that do not return void can be composed. |

**Function Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Element** | **Element Types** | **Occurrence Range** | **Description** |
| <Parameter> | Parameter | 0-unbounded | Represents the in and out parameters of this stored procedure. |
| <ElementType> | Type | 0-unbounded | Represents the tabular shapes the corresponding stored procedure can return. |
| <Return> | Return | 0-1 | The returned scalar type of this db function or stored procedure. If **Return** is null, the function returns void. A function cannot have both **Return** and **ElementType**. |

**TableFunction**

This element represents CUD override functions for tables. The LINQ to SQL designer allows creation of **Insert**, **Update**, and **Delete** override methods for LINQ TO SQL and allows mapping of entity property names to stored procedure parameter names.

The method name for CUD functions are fixed so there is no **Method** attribute in DBML for **TableFunction** elements. For example, for the Customer table, the CUD methods are named as **InsertCustomer**, **UpdateCustomer**, and **DeleteCustomer**.

A table function cannot return tabular shape so there is no **ElementType** attribute in **TableFunction** element.

**TableFunction Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | *(required)* | The name of the stored procedure within the database. |
| AccessModifier | AccessModifier | Private | The accessibility level of the stored procedure method. Valid values are **Public**, **Protected**, **Internal** and **Private**. |
| HasMultipleResults | Boolean | # of Types > 1 | Specifies if the stored procedure represented by this Function node returns multiple resultsets. Every resultset is a tabular shape, it can either be an existing **Type** or be a set of columns. In the latter case, a **Type** node will be created for the column set. |
| IsComposable | Boolean | False | Specifies if the function/stored procedure can be composed in LINQ to SQL queries. Only DB functions that do not return void can be composed. |

**TableFunction Sub-Element Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sub-Elements** | **Element Type** | **Occurrence Range** | **Description** |
| <Parameter> | TableFunctionParameter | 0-unbounded | Represents the in and out parameters of this table function. |
| <Return> | TableFunctionReturn | 0-1 | The returned scalar type of this table function. If **Return** is null, the function returns void. |

**Parameter**

This element represents a stored procedure/function parameter. Parameters can pass data in and out.

**Parameter Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Descriptions** |
| Name | String | *(required)* | The database name of the stored proc/function parameter. |
| Parameter | String | *Name* | The CLR name of the method parameter. |
|  | String | *(required)* | The CLR name of the method parameter. |
| DbType | String | *None* | The DB type of the stored proc/function parameter. |
| Direction | ParameterDirection | *In* | The direction that the parameter flows. Can be one of **In**, **Out**, and **InOut**. |

**Return**

This element represents the return type of a stored procedure/function.

**Return Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Type | String | *(required)* | The CLR type of the stored proc/function's result. |
| DbType | String | *None* | The DB type of the stored proc/function's result. |

**TableFunctionParameter**

This element represents a parameter of a CUD function. Parameters can pass data in and out. Every parameter is mapped to a **Table** column that this CUD function belongs to. There is no **Type** or **DbType** attributes in this element because type information can be obtained from the column to which the parameter maps.

**TableFunctionParameter Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| Name | String | *(required)* | The database name of the CUD function parameter. |
| Parameter | String | *Name* | The CLR name of the method parameter. |
| Column | String | *Name* | The column name this parameter is mapping to. |
| Direction | ParameterDirection | *In* | The direction that the parameter flows. Can be one of **In**, **Out**, or **InOut**. |
| Version | Version | Current | Whether **PropertyName** is referring to the current or original version of a given column. Only applicable during **Update** override. Can be **Current** or **Original**. |

**TableFunctionReturn**

This element represents a return type of a CUD function. It actually only contains the column name that is mapped to the result of the CUD function. The type information of the return can be obtained from the column.

**TableFunctionReturn Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attrobite** | **Type** | **Default** | **Description** |
| Column | String | *None* | The column name that the return is mapping to. |

**Connection**

This element represents default database **connection** parameters. This allows the creation of a default constructor for the **DataContext** type that already knows how to connect to a database.

There are two types of default connections possible, one with a direct **ConnectionString**, and one that reads from **App.Settings**.

**Connection Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Type** | **Default** | **Description** |
| UseApplicationSettings | Boolean | False | Determines whether to use an **App.Settings** file or get **Application** **Settings** from a direct **ConnectionString**. |
| ConnectionString | String | *None* | The connection string to send to the SQL data provider. |
| SettingsObjectName | String | Settings | The **App.Settings Object** to retrieve properties from. |
| SettingsPropertyName | String | ConnectionString | The **App.Settings** property that contains the **ConnectionString**. |

**Multi-tier Entities**

In two-tier applications, a single **DataContext** handles queries and updates. However, for applications with additional tiers, it is often necessary to use separate **DataContext** instances for query and updates. For example, in case of ASP.NET applications, query and update are done for separate requests to the Web server. Hence, it is impractical to use the same **DataContext** instance across multiple requests. In such cases, a **DataContext** instance needs to be able to update objects that it has not retrieved. The multi-tier entity support in LINQ to SQL provides such a capability through the **Attach()** method.

Here is an example of how a Customer object can be changed using a different **DataContext** instance:

**C#**

// Customer entity changed on another tier – for example, through a browser

// Back on the mid-tier, a new context needs to be used

Northwind db2 = new Northwind(…);

// Create a new entity for applying changes

Customer C2 = new Customer();

C2.CustomerID ="NewCustID";

// Set other properties needed for optimistic concurrency check

C2.CompanyName = "New Company Name Co.";

...

// Tell LINQ to SQL to track this object for an update; that is, not for insertion

db2.Customers.Attach(C2);

// Now apply the changes

C2.ContactName = "Mary Anders";

// DataContext now knows how to update the customer

db2.SubmitChanges();

**Visual Basic**

' Customer entity changed on another tier – for example, through a browser

' Back on the mid-tier, a new context needs to be used

Dim db2 As Northwind = New Northwind(…)

' Create a new entity for applying changes

Dim C2 As New Customer()

C2.CustomerID =”NewCustID”

' Set other properties needed for optimistic concurrency check

C2.CompanyName = ”New Company Name Co.”

...

' Tell LINQ to SQL to track this object for an update; that is, not for insertion

db2.Customers.Attach(C2)

' Now apply the changes

C2.ContactName = "Mary Anders"

' DataContext now knows how to update the customer

db2.SubmitChanges()

In multi-tier applications, the entire entity is often not sent across tiers for simplicity, interoperability, or privacy. For example, a supplier may define a data contract for a Web service that differs from the **Order** entity used on the middle tier. Likewise, a Web page may show only a subset of the members of an Employee entity. Hence, the multi-tier support is designed to accommodate such cases. Only the members belonging to one or more of the following categories need to be transported between tiers and set before calling **Attach()**.

1. Members that are part of the entity's identity.
2. Members that have been changed.
3. Members that participate in optimistic concurrency check.

If a timestamp or a version number column is used for optimistic concurrency check, then the corresponding member must be set before calling **Attach()**. Values for other members need not be set before calling **Attach()**. LINQ to SQL uses minimal updates with optimistic concurrency checks; that is, a member that is not set or checked for optimistic concurrency is ignored.

Original values required for optimistic concurrency checks may be retained using a variety of mechanisms outside the scope of LINQ to SQL APIs. An ASP.NET application may use a view state (or a control that uses the view state). A Web service may use the **DataContract** for an update method to ensure that the original values are available for update processing. In the interest of interoperability and generality, LINQ to SQL does not dictate the shape of the data exchanged between tiers or the mechanisms used for round-tripping the original values.

Entities for insertion and deletion do not require the **Attach()** method. The methods used for two-tier applications—**Table.Add()** and **Table.Remove()** can be used for insertion and deletion. As in case of two-tier updates, a user is responsible for handling foreign key constraints. A customer with orders cannot be just removed without handling its orders if there is a foreign key constraint in the database preventing the deletion of a customer with orders.

LINQ to SQL also handles attachment of entities for updates transitively. The user essentially creates the pre-update object graph as desired and calls **Attach()**. All changes can then be "replayed" on the attached graph to accomplish the necessary updates as shown below:

**C#**

Northwind db1 = new Northwind(…);

// Assume Customer c1 and related Orders o1, o2 are retrieved

// Back on the mid-tier, a new context needs to be used

Northwind db2 = new Northwind(…);

// Create new entities for applying changes

Customer c2 = new Customer();

c2.CustomerID = c.CustomerID;

Order o2 = new Order();

o2.OrderID = ...;

c2.Orders.Add(o2);

// Add other related objects needed for updates

// Set properties needed for optimistic concurrency check

...

// Order o1 to be deleted

Order o1 = new Order();

o1.OrderID = ...;

// Tell LINQ to SQL to track the graph transitively

db2.Customers.Attach(c2);

// Now "replay" all the changes

// Updates

c2.ContactName = ...;

o2.ShipAddress = ...;

// New object for insertion

Order o3 = new Order();

o3.OrderID = ...;

c2.Orders.Add(o3);

// Remove order o1

db2.Orders.Remove(o1);

// DataContext now knows how to do update/insert/delete

db2.SubmitChanges();

**Visual Basic**

Dim db1 As Northwind = New Northwind(…)

' Assume Customer c1 and related Orders o1, o2 are retrieved

' Back on the mid-tier, a new context needs to be used

Dim db2 As Northwind = New Northwind(…)

' Create new entities for applying changes

Customer c2 = new Customer()

c2.CustomerID = c.CustomerID

Dim o2 As Order = New Order()

o2.OrderID = ...

c2.Orders.Add(o2)

' Add other related objects needed for updates

' Set properties needed for optimistic concurrency check

...

' Order o1 to be deleted

Dim o1 As Order = New Order()

o1.OrderID = ...

' Tell LINQ to SQL to track the graph transitively

db2.Customers.Attach(c2)

' Now "replay" all the changes

' Updates

c2.ContactName = ...

o2.ShipAddress = ...

' New object for insertion

Dim o3 As Order = New Order()

o3.OrderID = ...

c2.Orders.Add(o3)

' Remove order o1

db2.Orders.Remove(o1)

' DataContext now knows how to do update/insert/delete

db2.SubmitChanges()

**External Mapping**

In addition to attribute-based mapping, LINQ to SQL also supports external mapping. The most common form of external mapping is an XML file. Mapping files enable additional scenarios where separating mapping from code is desirable.

**DataContext** provides an additional constructor for supplying a **MappingSource**. One form of **MappingSource** is an **XmlMappingSource** that can be constructed from an XML mapping file.

Here is an example of how mapping file can be used:

**C#**

String path = @"C:\Mapping\NorthwindMapping.xml";

XmlMappingSource prodMapping =

XmlMappingSource.FromXml(File.ReadAllText(path));

Northwind db = new Northwind(

@"Server=.\SQLExpress;Database=c:\Northwind\Northwnd.mdf",

prodMapping

);

**Visual Basic**

Dim path As String = "C:\Mapping\NorthwindMapping.xml"

Dim prodMapping As XmlMappingSource = \_

XmlMappingSource.FromXml(File.ReadAllText(path))

Dim db As Northwind = New Northwind( \_

"Server=.\SQLExpress;Database=c:\Northwind\Northwnd.mdf", \_

prodMapping )

Here is a corresponding snippet from the mapping file showing the mapping for **Product** class. It shows the class **Product** in namespace **Mapping** mapped to the **Products** table in **Northwind** database. The elements and attributes are consistent with the attribute names and parameters.

<?xml version="1.0" encoding="utf-8"?>

<Database xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance

xmlns:xsd="http://www.w3.org/2001/XMLSchema" Name="Northwind"

ProviderType="System.Data.Linq.SqlClient.Sql2005Provider">

<Table Name="Products">

<Type Name="Mappings.FunctionMapping.Product">

<Column Name="ProductID" Member="ProductID" Storage="\_ProductID"

DbType="Int NOT NULL IDENTITY" IsPrimaryKey="True"

IsDBGenerated="True" AutoSync="OnInsert" />

<Column Name="ProductName" Member="ProductName" Storage="\_ProductName"

DbType="NVarChar(40) NOT NULL" CanBeNull="False" />

<Column Name="SupplierID" Member="SupplierID" Storage="\_SupplierID"

DbType="Int" />

<Column Name="CategoryID" Member="CategoryID" Storage="\_CategoryID"

DbType="Int" />

<Column Name="QuantityPerUnit" Member="QuantityPerUnit"

Storage="\_QuantityPerUnit" DbType="NVarChar(20)" />

<Column Name="UnitPrice" Member="UnitPrice" Storage="\_UnitPrice"

DbType="Money" />

<Column Name="UnitsInStock" Member="UnitsInStock" Storage="\_UnitsInStock"

DbType="SmallInt" />

<Column Name="UnitsOnOrder" Member="UnitsOnOrder" Storage="\_UnitsOnOrder"

DbType="SmallInt" />

<Column Name="ReorderLevel" Member="ReorderLevel" Storage="\_ReorderLevel"

DbType="SmallInt" />

<Column Name="Discontinued" Member="Discontinued" Storage="\_Discontinued"

DbType="Bit NOT NULL" />

<Association Name="FK\_Order\_Details\_Products" Member="OrderDetails"

Storage="\_OrderDetails" ThisKey="ProductID" OtherTable="Order Details"

OtherKey="ProductID" DeleteRule="NO ACTION" />

<Association Name="FK\_Products\_Categories" Member="Category"

Storage="\_Category" ThisKey="CategoryID" OtherTable="Categories"

OtherKey="CategoryID" IsForeignKey="True" />

<Association Name="FK\_Products\_Suppliers" Member="Supplier"

Storage="\_Supplier" ThisKey="SupplierID" OtherTable="Suppliers"

OtherKey="SupplierID" IsForeignKey="True" />

</Type>

</Table>

</Database>

**NET Framework Function Support and Notes**

The following paragraphs provide basic information regarding LINQ to SQL type support and differences from the .NET Framework.

**Primitive Types**

**Implemented**

* Arithmetic and comparison operators
* Shift operators: **<< and >>**
* Conversion between char and numeric is done by **UNICODE**/**NCHAR**; otherwise SQL's **CONVERT** is used.

**Not implemented**

* **<Type>.Parse**
* Enums can be used and mapped to integers and strings in a table. For the latter, the **Parse** and **ToString()** methods are used.

**Difference from .NET**

* The output of **ToString** for double uses **CONVERT(NVARCHAR(30), @x, 2)** on SQL, which always uses 16 digits and "Scientific Notation." For example: "0.000000000000000e+000" for 0, so it does not give the same string as .NET's **Convert.ToString()**.

**System.String**

**Implemented**

* Non-static methods:
  + **Length**, **Substring**, **Contains**, **StartsWith**, **EndsWith**, **IndexOf**, **Insert**, **Remove**, **Replace**, **Trim**, **ToLower**, **ToUpper**, **LastIndexOf**, **PadRight**, **PadLeft**, **Equals**, **CompareTo**. All signatures are supported, except when they take the **StringComparison** parameter, and so on, as detailed below.
* Static methods:
* Concat(...) all signatures
* Compare(String, String)
* String (indexer)

Equals(String, String)

* Constructor:

String(Char, Int32)

* Operators:

+, ==, != (+, =, and <> in Visual Basic)

**Not implemented**

* Methods that take or produce an array of char.
* Methods that take a **CultureInfo**/**StringComparison**/**IFormatProvider**.
* Static (Shared in Visual Basic):
* Copy(String str)
* Compare(String, String, Boolean)
* Compare(String, String, StringComparison)
* Compare(String, String, Boolean, CultureInfo)
* Compare(String, Int32, String, Int32, Int32)
* Compare(String, Int32, String, Int32, Int32, Boolean)
* Compare(String, Int32, String, Int32, Int32, StringComparison)
* Compare(String, Int32, String, Int32, Int32, Boolean, CultureInfo)
* CompareOrdinal(String, String)
* CompareOrdinal(String, Int32, String, Int32, Int32)

Join(String, ArrayOf String [,...]) All Join version with first three args

* Instance:
* ToUpperInvariant()
* Format(String, Object) + overloads
* IndexOf(String, Int32, StringComparison)
* IndexOfAny(ArrayOf Char)
* Normalize()
* Normalize(NormalizationForm)
* IsNormalized()
* Split(...)
* StartsWith(String, StringComparison)
* ToCharArray()
* ToUpper(CultureInfo)
* TrimEnd(ParamArray Char)

TrimStart(ParamArray Char)

**Restrictions/Difference from .NET**

SQL uses collations to determine equality and ordering of strings. These can be specified on a SQL Server Instance, a database, a table column, or an expression.

The translations of the functions implemented so far do not change the collation or specify a different collation on the translated expressions. So if the default collation is case-insensitive, functions like **CompareTo** or **IndexOf** can give results that differ from what the (case sensitive) .NET functions would give.

The methods **StartsWith(str)**/**EndsWith(str)** assume the argument **str** is a constant or an expression that is evaluated on the client. That is, it is currently not possible to use a column for **str**.

**System.Math**

**Implemented static methods**

* All signatures:
  + **Abs**, **Acos**, **Asin**, **Atan**, **Atan2**, **BigMul**, **Ceiling**, **Cos**, **Cosh**, **Exp**, **Floor**, **Log**, **Log10**, **Max**, **Min**, **Pow**, **Sign**, **Sinh**, **Sqrt**, **Tan**, **Tanh**, or **Truncate**.

**Not implemented**

* **IEEERemainder**.
* **DivRem** has an out parameter, so you cannot use that in an expression. The constants **Math.PI** and **Math.E** are evaluated on the client, so they do not need a translation.

**Difference from .NET**

The translation of the .NET function **Math.Round** is the SQL function **ROUND**. The translation is supported only when an overload is specified that indicates the **MidpointRounding** enum value. The **MidpointRounding.AwayFromZero** is SQL behavior and **MidpointRounding.ToEven** indicates CLR behavior.

**System.Convert**

**Implemented**

* Methods of form **To<Type1>(<Type2> x)** where Type1, Type2 is one of:
  + **bool**, **byte**, **char**, **DateTime**, **decimal**, **double**, **float**, **Int16**, **Int32**, **Int64**, or **string**.
* The behavior is the same as a cast:
  + For **ToString(Double)** there is special code to get the full precision.
  + For conversion **Int32**/**Char**, LINQ to SQL uses SQL's **UNICODE**/**NCHAR** function.
  + Otherwise the translation is a **CONVERT**.

**Not Implemented**

* **ToSByte**, **UInt16**, **32**, **64**: These types do not exist in SQL.
* To<integer type>(String, Int32)
* ToString(..., Int32) any overload ending with an Int32 toBase
* IsDBNull(Object)
* GetTypeCode(Object)

ChangeType(...)

* Versions with the **IFormatProvider** parameter.
* Methods that involve an array (**To/FromBase64CharArray**, **To/FromBase64String**).

**System.TimeSpan**

**Implemented**

* Constructors:
* TimeSpan(Long)
* TimeSpan (year, month, day)
* TimeSpan (year, month, day, hour, minutes, seconds)

TimeSpan (year, month, day, hour, minutes, seconds, milliseconds)

* Operators:
* Comparison operators: <,==, and so on in C#; <, =, and so on in Visual Basic

+, -

* Static (Shared in Visual Basic) methods:

Compare(t1,t2)

* Non-static (Instance) methods / properties:
* Ticks, Milliseconds, Seconds, Hours, Days
* TotalMilliseconds, TotalSeconds, TotalMinutes, TotalHours, TotalDays,
* Equals, CompareTo(TimeSpan)
* Add(TimeSpan), Subtract(TimeSpan)

Duration() [= ABS], Negate()

**Not implemented**

ToString()

TimeSpan FromDay(Double), FromHours, all From Variants

TimeSpan Parse(String)

**System.DateTime**

**Implemented**

* Constructors:
* DateTime(year, month, day)
* DateTime(year, month, day, hour, minutes, seconds)

DateTime(year, month, day, hour, minutes, seconds, milliseconds)

* Operators:
* Comparisons
* DateTime – DateTime (gives TimeSpan)
* DateTime + TimeSpan (gives DateTime)

DateTime – TimeSpan (gives DateTime)

* Static (Shared) methods:
* Add(TimeSpan), AddTicks(Long),
* AddDays/Hours/Milliseconds/Minutes (Double)
* AddMonths/Years(Int32)

Equals

* Non-static (Instance) methods / properties:
* Day, Month, Year, Hour, Minute, Second, Millisecond, DayOfWeek
* CompareTo(DateTime)
* TimeOfDay()
* Equals

ToString()

**Difference from .NET**

SQL's datetime values are rounded to .000, .003 or .007 seconds, so it is less precise than those of .NET.

The range of SQL's datetime starts at January 1st, 1753.

SQL does not have a built-in type for **TimeSpan**. It uses different **DATEDIFF** methods that return 32-bit integers. One is **DATEDIFF(DAY,...)**, which gives the number of days; another is **DATEDIFF(MILLISECOND,...)**, which gives the number of milliseconds. An error results if the **DateTimes** are more than 24 days apart. In contrast, .NET uses 64-bit integers and measures **TimeSpans** in ticks.

To get as close as possible to the .NET semantics in SQL, LINQ to SQL translates **TimeSpans** into 64-bit integers and uses the two **DATEDIFF** methods mentioned above to calculate the number of ticks between two dates.

**DateTime** **UtcNow** is evaluated on the client when the query is translated (like any expression that does not involve database data).

**Not implemented**

IsDaylightSavingTime()

IsLeapYear(Int32)

DaysInMonth(Int32, Int32)

ToBinary()

ToFileTime()

ToFileTimeUtc()

ToLongDateString()

ToLongTimeString()

ToOADate()

ToShortDateString()

ToShortTimeString()

ToUniversalTime()

FromBinary(Long), FileTime, FileTimeUtc, OADate

GetDateTimeFormats(...)

constructor DateTime(Long)

Parse(String)

DayOfYear

**Debugging Support**

**DataContext** provides methods and properties to obtain the SQL generated for queries and change processing. These methods can be useful for understanding LINQ to SQL functionality and for debugging specific problems.

**DataContext Methods to Get Generated SQL**

|  |  |
| --- | --- |
| **Member** | **Purpose** |
| Log | Prints SQL before it is executed. Covers query, insert, update, delete commands. Usage:  **C#**  **db.Log = Console.Out;**  **Visual Basic**  **db.Log = Console.Out** |
| GetQueryText(query) | Returns the query text of the query without of executing it. Usage:  **C#**  **Console.WriteLine(db.GetQueryText(db.Customers));**  **Visual Basic**  **Console.WriteLine(db.GetQueryTest(db.Customers))** |
| GetChangeText() | Returns the text of SQL commands for insert/update/delete without executing them. Usage:  **C#**  **Console.WriteLine(db.GetChangeText());**  **Visual Basic**  **Console.WriteLine(db.GetChangeText())** |