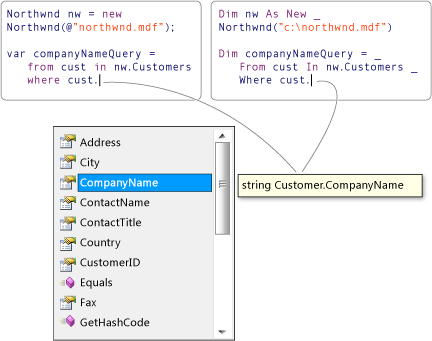
**Lecture 11 – LINQ**

Language integrated query. The new convenient abilities for collections manipulation.

Language-Integrated Query (LINQ) is a set of features introduced in Visual Studio 2008 that extends powerful query capabilities to the language syntax of C# and Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store. Visual Studio includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

Language-Integrated Query (LINQ) is an innovation introduced in Visual Studio 2008 and .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data.

Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. LINQ makes a *query* a first-class language construct in C# and Visual Basic. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The following illustration shows a partially-completed LINQ query against a SQL Server database in C# with full type checking and IntelliSense support.



In Visual Studio you can write LINQ queries in Visual Basic or C# with SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) or the generic [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx)interface. LINQ support for the ADO.NET Entity Framework is also planned, and LINQ providers are being written by third parties for many Web services and other database implementations.

You can use LINQ queries in new projects, or alongside non-LINQ queries in existing projects. The only requirement is that the project target .NET Framework 3.5 or later.

LINQ queries are based on generic types, which were introduced in version 2.0 of the .NET Framework. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1. When you create an instance of a generic collection class such as [List<T>](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx), you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as List<string>, and a list ofCustomer objects is expressed as List<Customer>. A generic list is strongly typed and provides many benefits over collections that store their elements as [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx). If you try to add a Customer to a List<string>, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
2. [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) is the interface that enables generic collection classes to be enumerated by using the **foreach** statement. Generic collection classes support [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) just as non-generic collection classes such as [ArrayList](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) support [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx).

[IEnumerable<T> variables in LINQ Queries](javascript:void(0))

LINQ query variables are typed as [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) or a derived type such as [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562.aspx). When you see a query variable that is typed as IEnumerable<Customer>, it just means that the query, when it is executed, will produce a sequence of zero or more Customer objects.

C#

IEnumerable<Customer> customerQuery =

from cust in customers

where cust.City == "London"

select cust;

foreach (Customer customer in customerQuery)

{

Console.WriteLine(customer.LastName + ", " + customer.FirstName);

}

[Letting the Compiler Handle Generic Type Declarations](javascript:void(0))

If you prefer, you can avoid generic syntax by using the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) keyword. The **var** keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the **from** clause. The following example produces the same compiled code as the previous example:

C#

var customerQuery2 =

from cust in customers

where cust.City == "London"

select cust;

foreach(var customer in customerQuery2)

{

Console.WriteLine(customer.LastName + ", " + customer.FirstName);

}

The **var** keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general, we recommend that if you use **var**, realize that it can make your code more difficult for others to read

* Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
* Query expressions are easy to master because they use many familiar C# language constructs.
* The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it.
* A query is not executed until you iterate over the query variable in a **foreach** statement.
* At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise.
* As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
* Some query operations, such as [Count](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count.aspx) or [Max](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.max.aspx), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query Syntax and Method Syntax in LINQ (C#)](http://msdn.microsoft.com/en-us/library/bb397947.aspx).
* Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) queries are compiled to delegates. [IQueryable](http://msdn.microsoft.com/en-us/library/system.linq.iqueryable.aspx) and [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562.aspx)queries are compiled to expression trees

[Obtaining a Data Source](javascript:void(0))

In a LINQ query, the first step is to specify the data source. In C# as in most programming languages a variable must be declared before it can be used. In a LINQ query, the **from** clause comes first in order to introduce the data source (customers) and the *range variable* (cust).

C#

//queryAllCustomers is an IEnumerable<Customer>

var queryAllCustomers = from cust in customers

select cust;

The range variable is like the iteration variable in a **foreach** loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable will serve as a reference to each successive element in customers. Because the compiler can infer the type of cust, you do not have to specify it explicitly. Additional range variables can be introduced by a **let** clause.

[Filtering](javascript:void(0))

Probably the most common query operation is to apply a filter in the form of a Boolean expression. The filter causes the query to return only those elements for which the expression is true. The result is produced by using the **where** clause. The filter in effect specifies which elements to exclude from the source sequence. In the following example, only those customers who have an address in London are returned.

C#

var queryLondonCustomers = from cust in customers

where cust.City == "London"

select cust;

You can use the familiar C# logical **AND** and **OR** operators to apply as many filter expressions as necessary in the **where** clause. For example, to return only customers from "London" **AND** whose name is "Devon" you would write the following code:

C#

where cust.City=="London" && cust.Name == "Devon"

To return customers from London or Paris, you would write the following code:

C#

where cust.City == "London" || cust.City == "Paris"

[Ordering](javascript:void(0))

Often it is convenient to sort the returned data. The **orderby** clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the Name property. Because Name is a string, the default comparer performs an alphabetical sort from A to Z.

C#

var queryLondonCustomers3 =

from cust in customers

where cust.City == "London"

orderby cust.Name ascending

select cust;

To order the results in reverse order, from Z to A, use the orderby…descending clause.

[Grouping](javascript:void(0))

The **group** clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the City so that all customers from London or Paris are in individual groups. In this case, cust.City is the key.

C#

// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>

var queryCustomersByCity =

from cust in customers

group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>

foreach (var customerGroup in queryCustomersByCity)

{

Console.WriteLine(customerGroup.Key);

foreach (Customer customer in customerGroup)

{

Console.WriteLine(" {0}", customer.Name);

}

}

When you end a query with a **group** clause, your results take the form of a list of lists. Each element in the list is an object that has a Key member and a list of elements that are grouped under that key. When you iterate over a query that produces a sequence of groups, you must use a nested **foreach** loop. The outer loop iterates over each group, and the inner loop iterates over each group's members.

If you must refer to the results of a group operation, you can use the **into** keyword to create an identifier that can be queried further. The following query returns only those groups that contain more than two customers:

C#

// custQuery is an IEnumerable<IGrouping<string, Customer>>

var custQuery =

from cust in customers

group cust by cust.City into custGroup

where custGroup.Count() > 2

orderby custGroup.Key

select custGroup;

[Joining](javascript:void(0))

Join operations create associations between sequences that are not explicitly modeled in the data sources. For example you can perform a join to find all the customers and distributors who have the same location. In LINQ the **join** clause always works against object collections instead of database tables directly.

C#

var innerJoinQuery =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorName = dist.Name };

In LINQ you do not have to use **join** as often as you do in SQL because foreign keys in LINQ are represented in the object model as properties that hold a collection of items. For example, a Customer object contains a collection of Order objects. Rather than performing a join, you access the orders by using dot notation:

from order in Customer.Orders...

[Selecting (Projections)](javascript:void(0))

The **select** clause produces the results of the query and specifies the "shape" or type of each returned element. For example, you can specify whether your results will consist of complete Customer objects, just one member, a subset of members, or some completely different result type based on a computation or new object creation. When the **select** clause produces something other than a copy of the source element, the operation is called a *projection*. The use of projections to transform data is a powerful capability of LINQ query expressions.

**LINQ TO SQL**

LINQ to SQL is a component of .NET Framework version 3.5 that provides a run-time infrastructure for managing relational data as objects.

In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When the application runs, LINQ to SQL translates into SQL the language-integrated queries in the object model and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back to objects that you can work with in your own programming language.

Developers using Visual Studio typically use the Object Relational Designer, which provides a user interface for implementing many of the features of LINQ to SQL.

The documentation that is included with this release of LINQ to SQL describes the basic building blocks, processes, and techniques you need for building LINQ to SQL applications.

By using LINQ to SQL, you can use the LINQ technology to access SQL databases just as you would access an in-memory collection.

For example, the nw object in the following code is created to represent the Northwind database, the Customers table is targeted, the rows are filtered for Customers from London, and a string for CompanyName is selected for retrieval.

When the loop is executed, the collection of CompanyName values is retrieved.

C#

// Northwnd inherits from System.Data.Linq.DataContext.

Northwnd nw = new Northwnd(@"northwnd.mdf");

// or, if you are not using SQL Server Express

// Northwnd nw = new Northwnd("Database=Northwind;Server=server\_name;Integrated Security=SSPI");

var companyNameQuery =

from cust in nw.Customers

where cust.City == "London"

select cust.CompanyName;

foreach (var customer in companyNameQuery)

{

Console.WriteLine(customer);

}

To execute a SQL **Insert**, just add objects to the object model you have created, and call [SubmitChanges](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.submitchanges.aspx) on the [DataContext](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.aspx).

In the following example, a new customer and information about the customer is added to the Customers table by using [InsertOnSubmit](http://msdn.microsoft.com/en-us/library/bb763516.aspx).

C#

// Northwnd inherits from System.Data.Linq.DataContext.

Northwnd nw = new Northwnd(@"northwnd.mdf");

Customer cust = new Customer();

cust.CompanyName = "SomeCompany";

cust.City = "London";

cust.CustomerID = "98128";

cust.PostalCode = "55555";

cust.Phone = "555-555-5555";

nw.Customers.InsertOnSubmit(cust);

// At this point, the new Customer object is added in the object model.

// In LINQ to SQL, the change is not sent to the database until

// SubmitChanges is called.

nw.SubmitChanges();

[Updating](javascript:void(0))

To **Update** a database entry, first retrieve the item and edit it directly in the object model. After you have modified the object, call [SubmitChanges](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.submitchanges.aspx) on the [DataContext](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.aspx) to update the database.

In the following example, all customers who are from London are retrieved. Then the name of the city is changed from "London" to "London - Metro". Finally, [SubmitChanges](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.submitchanges.aspx) is called to send the changes to the database.

C#

Northwnd nw = new Northwnd(@"northwnd.mdf");

var cityNameQuery =

from cust in nw.Customers

where cust.City.Contains("London")

select cust;

foreach (var customer in cityNameQuery)

{

if (customer.City == "London")

{

customer.City = "London - Metro";

}

}

nw.SubmitChanges();

[Deleting](javascript:void(0))

To **Delete** an item, remove the item from the collection to which it belongs, and then call [SubmitChanges](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.submitchanges.aspx) on the [DataContext](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.aspx) to commit the change.

In the following example, the customer who has CustomerID of 98128 is retrieved from the database. Then, after confirming that the customer row was retrieved, [DeleteOnSubmit](http://msdn.microsoft.com/en-us/library/bb763473.aspx) is called to remove that object from the collection. Finally, [SubmitChanges](http://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.submitchanges.aspx) is called to forward the deletion to the database.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/bb882643.aspx?cs-save-lang=1&cs-lang=vb#code-snippet-4)

Northwnd nw = new Northwnd(@"northwnd.mdf");

var deleteIndivCust =

from cust in nw.Customers

where cust.CustomerID == "98128"

select cust;

if (deleteIndivCust.Count() > 0)

{

nw.Customers.DeleteOnSubmit(deleteIndivCust.First());

nw.SubmitChanges();

}

**Questions**

1. What is LINQ?
2. How LINQ simplifies the work with collections?
3. What kind of operations LINQ provide?
4. How link concern the Lambdas?