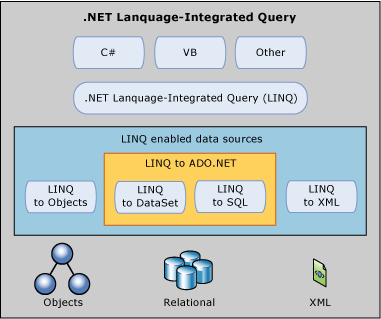
.NET Framework Developer's Guide

**LINQ to ADO.NET**

Language-Integrated Query (LINQ) defines a set of general purpose standard query operators that you can use in .NET Framework 3.0 programming languages. These standard query operators enable you to project, filter, and traverse in-memory collections or tables in a database. Note that the LINQ queries are expressed in the programming language itself, and not as string literals embedded in the application code. This is a significant change from the way most applications have been written on earlier versions of the .NET Framework. Writing queries from within your programming language offers several key advantages. It simplifies querying by eliminating the need to use a separate query language. And if you use the Visual Studio 2008 IDE, LINQ also lets you take advantage of compile-time checking, static typing, and IntelliSense.

LINQ is integrated into various aspects of data access in the .NET Framework, including the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset(VS.90).aspx) disconnected programming model and existing SQL Server database schemas. This section describes LINQ to ADO.NET, the ADO.NET implementation of LINQ.

The following diagram provides an overview of how LINQ to ADO.NET relates to high-level programming languages, other LINQ technologies, and LINQ-enabled data sources.



.NET Framework 4

**LINQ and ADO.NET**

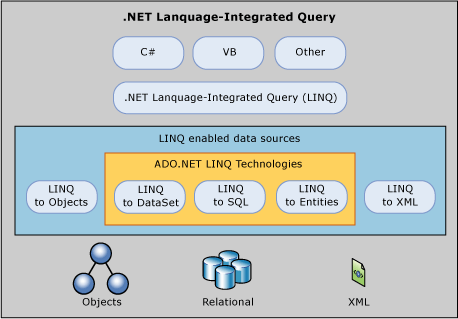
Today, many business developers must use two (or more) programming languages: a high-level language for the business logic and presentation layers (such as Visual C# or Visual Basic), and a query language to interact with the database (such as Transact-SQL). This requires the developer to be proficient in several languages to be effective, and also causes language mismatches in the development environment. For example, an application that uses a data access API to execute a query against a database specifies the query as a string literal by using quotation marks. This query string is un-readable to the compiler and is not checked for errors, such as invalid syntax or whether the columns or rows it references actually exist. There is no type checking of the query parameters and no **IntelliSense** support, either.

Language-Integrated Query (LINQ) enables developers to form set-based queries in their application code, without having to use a separate query language. You can write LINQ queries against various enumerable data sources (that is, a data source that implements the [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) interface), such as in-memory data structures, XML documents, SQL databases, and [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) objects. Although these enumerable data sources are implemented in various ways, they all expose the same syntax and language constructs. Because queries can be formed in the programming language itself, you do not have to use another query language that is embedded as string literals that cannot be understood or verified by the compiler. Integrating queries into the programming language also enables Visual Studio programmers to be more productive by providing compile-time type and syntax checking, and **IntelliSense**. These features reduce the need for query debugging and error fixing.

Transferring data from SQL tables into objects in memory is often tedious and error-prone. The LINQ provider implemented by LINQ to DataSet and LINQ to SQL converts the source data into [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx)-based object collections. The programmer always views the data as an [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) collection, both when you query and when you update. Full **IntelliSense** support is provided for writing queries against those collections.

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) and LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

The following diagram provides an overview of how the ADO.NET LINQ technologies relate to high-level programming languages and LINQ-enabled data sources.



For general information on the LINQ language features, see [Introduction to LINQ](http://msdn.microsoft.com/en-us/library/bb397897.aspx). For information about using LINQ in your applications, see the [LINQ General Programming Guide](http://msdn.microsoft.com/en-us/library/bb397912.aspx), which contains detailed information about how to use LINQ technologies.

The following sections provide more information about LINQ to DataSet, LINQ to SQL, and LINQ to Entities.

http://i.msdn.microsoft.com/Global/Images/clear.gif LINQ to DataSet

The [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) is a key element of the disconnected programming model that ADO.NET is built on, and is widely used. LINQ to DataSet enables developers to build richer query capabilities into [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) by using the same query formulation mechanism that is available for many other data sources. For more information, see [LINQ to DataSet](http://msdn.microsoft.com/en-us/library/bb386977.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif LINQ to SQL

LINQ to SQL is a useful tool for developers who do not require mapping to a conceptual model. By using LINQ to SQL, you can use the LINQ programming model directly over existing database schema. LINQ to SQL enables developers to generate .NET Framework classes that represent data. Rather than mapping to a conceptual data model, these generated classes map directly to database tables, views, stored procedures, and user-defined functions.

With LINQ to SQL, developers can write code directly against the storage schema using the same LINQ programming pattern as in-memory collections and the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx), in addition to other data sources such as XML. For more information, see [LINQ to SQL](http://msdn.microsoft.com/en-us/library/bb386976.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif LINQ to Entities

Most applications are currently written on top of relational databases. At some point, these applications will need to interact with the data represented in a relational form. Database schemas are not always ideal for building applications, and the conceptual models of application are not the same as the logical models of databases. The Entity Data Model is a conceptual data model that can be used to model the data of a particular domain so that applications can interact with data as objects. See [ADO.NET Entity Framework](http://msdn.microsoft.com/en-us/library/bb399572.aspx) for more information.

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities. See [LINQ to Entities](http://msdn.microsoft.com/en-us/library/bb386964.aspx) for more information.

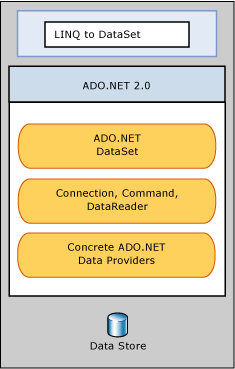
.NET Framework 4

**LINQ to DataSet**

LINQ to DataSet makes it easier and faster to query over data cached in a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) object. Specifically, LINQ to DataSet simplifies querying by enabling developers to write queries from the programming language itself, instead of by using a separate query language. This is especially useful for Visual Studio developers, who can now take advantage of the compile-time syntax checking, static typing, and IntelliSense support provided by the Visual Studio in their queries.

LINQ to DataSet can also be used to query over data that has been consolidated from one or more data sources. This enables many scenarios that require flexibility in how data is represented and handled, such as querying locally aggregated data and middle-tier caching in Web applications. In particular, generic reporting, analysis, and business intelligence applications require this method of manipulation.

The LINQ to DataSet functionality is exposed primarily through the extension methods in the [DataRowExtensions](http://msdn.microsoft.com/en-us/library/system.data.datarowextensions.aspx) and [DataTableExtensions](http://msdn.microsoft.com/en-us/library/system.data.datatableextensions.aspx) classes. LINQ to DataSet builds on and uses the existing ADO.NET 2.0 architecture, and is not meant to replace ADO.NET 2.0 in application code. Existing ADO.NET 2.0 code will continue to function in a LINQ to DataSet application. The relationship of LINQ to DataSet to ADO.NET 2.0 and the data store is illustrated in the following diagram.



LINQ to SQL

**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.

|  |
| --- |
| **NoteNote** |
| Relational data appears as a collection of two-dimensional tables (*relations* or *flat files*), where common columns relate tables to each other. To use LINQ to SQL effectively, you must have some familiarity with the underlying principles of relational databases. |

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. For more information, see [Object Relational Designer (O/R Designer)](http://msdn.microsoft.com/en-us/library/Bb384429(en-us,VS.100).aspx).

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. You can also search the MSDN Library for specific issues, and you can participate in the [LINQ Forum](http://go.microsoft.com/fwlink/?LinkId=76488), where you can discuss more complex topics in detail with experts. Finally, the [LINQ to SQL: .NET Language-Integrated Query for Relational Data](http://go.microsoft.com/fwlink/?LinkId=93205) white paper details LINQ to SQL technology, complete with Visual Basic and C# code examples.

LINQ to SQL

**Getting Started (LINQ to SQL)**

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.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

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

Dim nw As New Northwnd("c:\northwnd.mdf")

' or, if you are not using SQL Server Express

' Dim nw As New Northwnd("Database=Northwind;Server=dschwart7;Integrated Security=SSPI")

Dim companyNameQuery = \_

From cust In nw.Customers \_

Where cust.City = "London" \_

Select cust.CompanyName

For Each customer In companyNameQuery

Console.WriteLine(customer)

Next

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl02_code');" \o "Copy Code)

// 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);

}

**LINQ to Entities**

LINQ to Entities provides Language-Integrated Query (LINQ) support that enables developers to write queries against the Entity Framework conceptual model using Visual Basic or Visual C#. Queries against the Entity Framework are represented by command tree queries, which execute against the object context. LINQ to Entities converts Language-Integrated Queries (LINQ) queries to command tree queries, executes the queries against the Entity Framework, and returns objects that can be used by both the Entity Framework and LINQ. The following is the process for creating and executing a LINQ to Entities query:

1. Construct an [ObjectQuery](http://msdn.microsoft.com/en-us/library/bb345303.aspx) instance from [ObjectContext](http://msdn.microsoft.com/en-us/library/system.data.objects.objectcontext.aspx).
2. Compose a LINQ to Entities query in C# or Visual Basic by using the **ObjectQuery** instance.
3. Convert LINQ standard query operators and expressions to command trees.
4. Execute the query, in command tree representation, against the data source. Any exceptions thrown on the data source during execution are passed directly up to the client.
5. Return query results back to the client.

**Constructing an ObjectQuery Instance**

The **ObjectQuery** generic class represents a query that returns a collection of zero or more typed entities. An object query is typically constructed from an existing object context, instead of being manually constructed, and always belongs to that object context. This context provides the connection and metadata information that is required to compose and execute the query. The **ObjectQuery** generic class implements the [IQueryable](http://msdn.microsoft.com/en-us/library/bb351562.aspx) generic interface, whose builder methods enable LINQ queries to be incrementally built. You can also let the compiler infer the type of entities by using the C# **var** keyword (**Dim** in Visual Basic, with local type inference enabled).

**Composing the Queries**

Instances of the **ObjectQuery** generic class, which implements the generic **IQueryable** interface, serve as the data source for LINQ to Entities queries. In a query, you specify exactly the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. In LINQ, a query is stored in a variable. This query variable takes no action and returns no data; it only stores the query information. After you create a query you must execute that query to retrieve any data.

LINQ to Entities queries can be composed in two different syntaxes: query expression syntax and method-based query syntax. Query expression syntax and method-based query syntax are new in C# 3.0 and Visual Basic 9.0.

For more information, see [Queries in LINQ to Entities](http://msdn.microsoft.com/en-us/library/bb399367.aspx).

**Query Conversion**

To execute a LINQ to Entities query against the Entity Framework, the LINQ query must be converted to a command tree representation that can be executed against the Entity Framework.

LINQ to Entities queries are comprised of LINQ standard query operators (such as [Select](http://msdn.microsoft.com/en-us/library/system.linq.queryable.select.aspx), [Where](http://msdn.microsoft.com/en-us/library/system.linq.queryable.where.aspx), and [GroupBy](http://msdn.microsoft.com/en-us/library/system.linq.queryable.groupby.aspx)) and expressions (x > 10, Contact.LastName, and so on). LINQ operators are not defined by a class, but rather are methods on a class. In LINQ, expressions can contain anything allowed by types within the [System.Linq.Expressions](http://msdn.microsoft.com/en-us/library/system.linq.expressions.aspx) namespace and, by extension, anything that can be represented in a lambda function. This is a superset of the expressions that are allowed by the Entity Framework, which are by definition restricted to operations allowed on the database, and supported by **ObjectQuery**.

In the Entity Framework, both operators and expressions are represented by a single type hierarchy, which are then placed in a command tree. The command tree is used by the Entity Framework to execute the query. If the LINQ query cannot be expressed as a command tree, an exception will be thrown when the query is being converted. The conversion of LINQ to Entities queries involves two sub-conversions: the conversion of the standard query operators, and the conversion of the expressions.

There are a number of LINQ standard query operators that do not have a valid translation in LINQ to Entities. Attempts to use these operators will result in an exception at query translation time. For a list of supported LINQ to Entities operators, see [Supported and Unsupported LINQ Methods (LINQ to Entities)](http://msdn.microsoft.com/en-us/library/bb738550.aspx).

For more information about using the standard query operators in LINQ to Entities, see [Standard Query Operators in LINQ to Entities Queries](http://msdn.microsoft.com/en-us/library/bb738551.aspx).

In general, expressions in LINQ to Entities are evaluated on the server, so the behavior of the expression should not be expected to follow CLR semantics. For more information, see [Expressions in LINQ to Entities Queries](http://msdn.microsoft.com/en-us/library/bb738645.aspx).

For information about how CLR method calls are mapped to canonical functions in the data source, see [CLR Method to Canonical Function Mapping](http://msdn.microsoft.com/en-us/library/bb738681.aspx).

For information about how to call canonical, database, and custom functions from within LINQ to Entities queries, see [Calling Functions in LINQ to Entities Queries](http://msdn.microsoft.com/en-us/library/dd456828.aspx).

**Query Execution**

After the LINQ query is created by the user, it is converted to a representation that is compatible with the Entity Framework (in the form of command trees), which is then executed against the data source. At query execution time, all query expressions (or components of the query) are evaluated on the client or on the server. This includes expressions that are used in result materialization or entity projections. For more information, see [Query Execution](http://msdn.microsoft.com/en-us/library/bb738633.aspx). For information on how to improve performance by compiling a query once and then executing it several times with different parameters, see [Compiled Queries (LINQ to Entities)](http://msdn.microsoft.com/en-us/library/bb896297.aspx).

**Materialization**

Materialization is the process of returning query results back to the client as CLR types. In LINQ to Entities, query results data records are never returned; there is always a backing CLR type, defined by the user or by the Entity Framework, or generated by the compiler (anonymous types). All object materialization is performed by the Entity Framework. Any errors that result from an inability to map between the Entity Framework and the CLR will cause exceptions to be thrown during object materialization.

Query results are usually returned as one of the following:

* A collection of zero or more typed entity objects or a projection of complex types defined in the conceptual model.
* CLR types that are supported by the Entity Framework.
* Inline collections.
* Anonymous types.