Language-Integrated Query (LINQ) (Visual Basic)

Visual Studio 2015

LINQ is a set of features that extends powerful query capabilities to the language syntax of 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. The .NET Framework includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

In This Section

Introduction to LINQ (Visual Basic)
: Provides a general introduction to the kinds of applications that you can write and the kinds of problems that you can solve with LINQ queries.

Getting Started with LINQ in Visual Basic
: Describes the basic facts you should know in order to understand the Visual Basic documentation and samples.

Visual Studio IDE and Tools Support for LINQ (Visual Basic)
: Describes Visual Studio's Object Relational Designer, debugger support for queries, and other IDE features related to LINQ.

Standard Query Operators Overview (Visual Basic)
: Provides an introduction to the standard query operators. It also provides links to topics that have more information about each type of query operation.

LINQ to Objects (Visual Basic)
: Includes links to topics that explain how to use LINQ to Objects to access in-memory data structures,

LINQ to XML (Visual Basic)
: Includes links to topics that explain how to use LINQ to XML, which provides the in-memory document modification capabilities of the Document Object Model (DOM), and supports LINQ query expressions.

LINQ to ADO.NET (Portal Page)
: Provides an entry point for documentation about LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet enables you to build richer query capabilities into DataSet by using the same query functionality that is available for other data sources. LINQ to SQL provides a run-time infrastructure for managing relational data as objects. LINQ to Entities enables developers to write queries against the Entity Framework conceptual model by using C#.

Enabling a Data Source for LINQ Querying
: Provides an introduction to custom LINQ providers, LINQ expression trees, and other ways to extend LINQ.

LINQ to ADO.NET (Portal Page)

Visual Studio 2015

LINQ to ADO.NET enables you to query over any enumerable object in ADO.NET by using the Language-Integrated Query (LINQ) programming model.

> **Note**
>
> The LINQ to ADO.NET documentation is located in the ADO.NET section of the .NET Framework SDK: LINQ and ADO.NET.

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, LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

LINQ to DataSet

The DataSet is one of the most widely used components in ADO.NET, and is a key element of the disconnected programming model that ADO.NET is built on. Despite this prominence, however, the DataSet has limited query capabilities.

LINQ to DataSet enables you to build richer query capabilities into DataSet by using the same query functionality that is available for many other data sources.

For more information, see LINQ to DataSet.

LINQ to SQL

LINQ to SQL 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 you execute the application, LINQ to SQL translates language-integrated queries in the object model into SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back into objects that you can manipulate.

LINQ to SQL includes support for stored procedures and user-defined functions in the database, and for inheritance in the object model.

For more information, see LINQ to SQL.

LINQ to Entities

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 for more information.

See Also

LINQ and ADO.NET
Language-Integrated Query (LINQ) (Visual Basic)

LINQ to Entities

.NET Framework (current version)

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(Of T) instance from ObjectContext.
2. Compose a LINQ to Entities query in C# or Visual Basic by using the ObjectQuery(Of T) 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(Of T) 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(Of T) generic class implements the IQueryable(Of T) 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(Of T) generic class, which implements the generic IQueryable(Of T) 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.

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, Where, and GroupBy) 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 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(Of T).

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).

For more information about using the standard query operators in LINQ to Entities, see Standard Query Operators in LINQ to Entities Queries.

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.

For information about how CLR method calls are mapped to canonical functions in the data source, see CLR Method to Canonical Function Mapping.

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.

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. 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).

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.

For more information, see Query Results.

In This Section

Queries in LINQ to Entities

Expressions in LINQ to Entities Queries

Calling Functions in LINQ to Entities Queries

Compiled Queries (LINQ to Entities)

Query Execution

Query Results

Standard Query Operators in LINQ to Entities Queries

CLR Method to Canonical Function Mapping

Supported and Unsupported LINQ Methods (LINQ to Entities)

Known Issues and Considerations in LINQ to Entities

See Also

Known Issues and Considerations in LINQ to Entities
LINQ (Language-Integrated Query)
LINQ and ADO.NET
ADO.NET Entity Framework

Entity SQL Reference

.NET Framework (current version)

This section contains Entity SQL reference topics. This topic summarizes and groups the Entity SQL operators by category.

Arithmetic Operators

Arithmetic operators perform mathematical operations on two expressions of one or more numeric data types. The following table lists the Entity SQL arithmetic operators.

Operator	Use
+ (Add)	Addition.
/ (Divide)	Division.
% (Modulo)	Returns the remainder of a division.
* (Multiply)	Multiplication.
- (Negative)	Negation.
- (Subtract)	Subtraction.

Canonical Functions

Canonical functions are supported by all data providers and can be used by all querying technologies. The following table lists the canonical functions.

Function	Type
Aggregate Entity SQL Canonical Functions	Discusses aggregate Entity SQL canonical functions.
Math Canonical Functions	Discusses math Entity SQL canonical functions.
String Canonical Functions	Discusses string Entity SQL canonical functions.
Date and Time Canonical Functions	Discusses date and time Entity SQL canonical functions.

Bitwise Canonical Functions	Discusses bitwise Entity SQL canonical functions.
Other Canonical Functions	Discusses functions not classified as bitwise, date/time, string, math, or aggregate.

Comparison Operators

Comparison operators are defined for the following types: **Byte**, **Int16**, **Int32**, **Int64**, **Double**, **Single**, **Decimal**, **String**, **DateTime**, **Date**, **Time**, **DateTimeOffset**. Implicit type promotion occurs for the operands before the comparison operator is applied. Comparison operators always yield Boolean values. When at least one of the operands is **null**, the result is **null**.

Equality and inequality are defined for any object type that has identity, such as the **Boolean** type. Non-primitive objects with identity are considered equal if they share the same identity. The following table lists the Entity SQL comparison operators.

Operator	Description
= (Equals)	Compares the equality of two expressions.
> (Greater Than)	Compares two expressions to determine whether the left expression has a value greater than the right expression.
>= (Greater Than or Equal To)	Compares two expressions to determine whether the left expression has a value greater than or equal to the right expression.
IS [NOT] NULL	Determines if a query expression is null.
< (Less Than)	Compares two expressions to determine whether the left expression has a value less than the right expression.
<= (Less Than or Equal To)	Compares two expressions to determine whether the left expression has a value less than or equal to the right expression.
[NOT] BETWEEN	Determines whether an expression results in a value in a specified range.
!= (Not Equal To)	Compares two expressions to determine whether the left expression is not equal to the right expression.
[NOT] LIKE	Determines whether a specific character string matches a specified pattern.

Logical and Case Expression Operators

Logical operators test for the truth of a condition. The CASE expression evaluates a set of Boolean expressions to determine the result. The following table lists the logical and CASE expression operators.

Operator	Description
&& (Logical AND)	Logical AND.
! (Logical NOT)	Logical NOT.
|| (Logical OR)	Logical OR.
CASE	Evaluates a set of Boolean expressions to determine the result.
THEN	The result of a WHEN clause when it evaluates to true.

Query Operators

Query operators are used to define query expressions that return entity data. The following table lists query operators.

Operator	Use
FROM	Specifies the collection that is used in SELECT statements.
GROUP BY	Specifies groups into which objects that are returned by a query (SELECT) expression are to be placed.
GroupPartition	Returns a collection of argument values, projected off the group partition to which the aggregate is related.
HAVING	Specifies a search condition for a group or an aggregate.
LIMIT	Used with the ORDER BY clause to performed physical paging.
ORDER BY	Specifies the sort order that is used on objects returned in a SELECT statement.
SELECT	Specifies the elements in the projection that are returned by a query.
SKIP	Used with the ORDER BY clause to performed physical paging.
TOP	Specifies that only the first set of rows will be returned from the query result.
WHERE	Conditionally filters data that is returned by a query.

Reference Operators

A reference is a logical pointer (foreign key) to a specific entity in a specific entity set. Entity SQL supports the following operators to construct, deconstruct, and navigate through references.

Operator	Use
CREATEREF	Creates references to an entity in an entity set.
DEREF	Dereferences a reference value and produces the result of that dereference.
KEY	Extracts the key of a reference or of an entity expression.
NAVIGATE	Allows you to navigate over the relationship from one entity type to another
REF	Returns a reference to an entity instance.

Set Operators

Entity SQL provides various powerful set operations. This includes set operators similar to Transact-SQL operators such as UNION, INTERSECT, EXCEPT, and EXISTS. Entity SQL also supports operators for duplicate elimination (SET), membership testing (IN), and joins (JOIN). The following table lists the Entity SQL set operators.

Operator	Use
ANYELEMENT	Extracts an element from a multivalued collection.
EXCEPT	Returns a collection of any distinct values from the query expression to the left of the EXCEPT operand that are not also returned from the query expression to the right of the EXCEPT operand.
[NOT] EXISTS	Determines if a collection is empty.
FLATTEN	Converts a collection of collections into a flattened collection.
[NOT] IN	Determines whether a value matches any value in a collection.
INTERSECT	Returns a collection of any distinct values that are returned by both the query expressions on the left and right sides of the INTERSECT operand.
OVERLAPS	Determines whether two collections have common elements.
SET	Used to convert a collection of objects into a set by yielding a new collection with all duplicate elements removed.

UNION	Combines the results of two or more queries into a single collection.

Type Operators

Entity SQL provides operations that allow the type of an expression (value) to be constructed, queried, and manipulated. The following table lists operators that are used to work with types.

Operator	Use
CAST	Converts an expression of one data type to another.
COLLECTION	Used in a FUNCTION operation to declare a collection of entity types or complex types.
IS [NOT] OF	Determines whether the type of an expression is of the specified type or one of its subtypes.
OFTYPE	Returns a collection of objects from a query expression that is of a specific type.
Named Type Constructor	Used to create instances of entity types or complex types.
MULTISET	Creates an instance of a multiset from a list of values.
ROW	Constructs anonymous, structurally typed records from one or more values.
TREAT	Treats an object of a particular base type as an object of the specified derived type.

Other Operators

The following table lists other Entity SQL operators.

Operator	Use
+ (String Concatenation)	Used to concatenate strings in Entity SQL.
. (Member Access)	Used to access the value of a property or field of an instance of structural conceptual model type.
-- (Comment)	Include Entity SQL comments.

FUNCTION	Defines an inline function that can be executed in an Entity SQL query.

See Also

Entity SQL Language

LINQ and Strings (Visual Basic)

Visual Studio 2015

LINQ can be used to query and transform strings and collections of strings. It can be especially useful with semi-structured data in text files. LINQ queries can be combined with traditional string functions and regular expressions. For example, you can use the Split or Split method to create an array of strings that you can then query or modify by using LINQ. You can use the IsMatch method in the **where** clause of a LINQ query. And you can use LINQ to query or modify the MatchCollection results returned by a regular expression.

You can also use the techniques described in this section to transform semi-structured text data to XML. For more information, see How to: Generate XML from CSV Files.

The examples in this section fall into two categories:

Querying a Block of Text

You can query, analyze, and modify text blocks by splitting them into a queryable array of smaller strings by using the Split method or the Split method. You can split the source text into words, sentences, paragraphs, pages, or any other criteria, and then perform additional splits if they are required in your query.

How to: Count Occurrences of a Word in a String (LINQ) (Visual Basic)
: Shows how to use LINQ for simple querying over text.

How to: Query for Sentences that Contain a Specified Set of Words (LINQ) (Visual Basic)
: Shows how to split text files on arbitrary boundaries and how to perform queries against each part.

How to: Query for Characters in a String (LINQ) (Visual Basic)
: Demonstrates that a string is a queryable type.

How to: Combine LINQ Queries with Regular Expressions (Visual Basic)
: Shows how to use regular expressions in LINQ queries for complex pattern matching on filtered query results.

Querying Semi-Structured Data in Text Format

Many different types of text files consist of a series of lines, often with similar formatting, such as tab- or comma-delimited files or fixed-length lines. After you read such a text file into memory, you can use LINQ to query and/or modify the lines. LINQ queries also simplify the task of combining data from multiple sources.

How to: Find the Set Difference Between Two Lists (LINQ) (Visual Basic)
: Shows how to find all the strings that are present in one list but not the other.

How to: Sort or Filter Text Data by Any Word or Field (LINQ) (Visual Basic)
: Shows how to sort text lines based on any word or field.

How to: Reorder the Fields of a Delimited File (LINQ) (Visual Basic)

Shows how to reorder fields in a line in a .csv file.

How to: Combine and Compare String Collections (LINQ) (Visual Basic)
Shows how to combine string lists in various ways.

How to: Populate Object Collections from Multiple Sources (LINQ) (Visual Basic)
Shows how to create object collections by using multiple text files as data sources.

How to: Join Content from Dissimilar Files (LINQ) (Visual Basic)
Shows how to combine strings in two lists into a single string by using a matching key.

How to: Split a File Into Many Files by Using Groups (LINQ) (Visual Basic)
Shows how to create new files by using a single file as a data source.

How to: Compute Column Values in a CSV Text File (LINQ) (Visual Basic)
Shows how to perform mathematical computations on text data in .csv files.

See Also

Language-Integrated Query (LINQ) (Visual Basic)
How to: Generate XML from CSV Files

LINQ and File Directories (Visual Basic)

Visual Studio 2015

Many file system operations are essentially queries and are therefore well-suited to the LINQ approach.

Note that the queries in this section are non-destructive. They are not used to change the contents of the original files or folders. This follows the rule that queries should not cause any side-effects. In general, any code (including queries that perform create / update / delete operators) that modifies source data should be kept separate from the code that just queries the data.

This section contains the following topics:

How to: Query for Files with a Specified Attribute or Name (Visual Basic)
: Shows how to search for files by examining one or more properties of its FileInfo object.

How to: Group Files by Extension (LINQ) (Visual Basic)
: Shows how to return groups of FileInfo object based on their file name extension.

How to: Query for the Total Number of Bytes in a Set of Folders (LINQ) (Visual Basic)
: Shows how to return the total number of bytes in all the files in a specified directory tree.

How to: Compare the Contents of Two Folders (LINQ) (Visual Basic)s
: Shows how to return all the files that are present in two specified folders, and also all the files that are present in one folder but not the other.

How to: Query for the Largest File or Files in a Directory Tree (LINQ) (Visual Basic)
: Shows how to return the largest or smallest file, or a specified number of files, in a directory tree.

How to: Query for Duplicate Files in a Directory Tree (LINQ) (Visual Basic)
: Shows how to group for all file names that occur in more than one location in a specified directory tree. Also shows how to perform more complex comparisons based on a custom comparer.

How to: Query the Contents of Files in a Folder (LINQ) (Visual Basic)
: Shows how to iterate through folders in a tree, open each file, and query the file's contents.

Comments

There is some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of FileInfo objects that represents all the files under a specified root folder and all its subfolders. The actual state of each FileInfo may change in the time between when you begin and end executing a query. For example, you can create a list of FileInfo objects to use as a data source. If you try to access the **Length** property in a query, the FileInfo object will try to access the file system to update the value of **Length**. If the file no longer exists, you will get a FileNotFoundException in your query, even though you are not querying the file system directly. Some queries in this section use a separate method that consumes these particular exceptions in certain cases. Another option is to keep your data source updated dynamically by using the FileSystemWatcher.

See Also

LINQ to Objects (Visual Basic)

How to: Query An Assembly's Metadata with Reflection (LINQ) (Visual Basic)

Visual Studio 2015

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

VB

```
Imports System.Reflection
Imports System.IO
Imports System.Linq
Module Module1

    Sub Main()
        Dim asmbly As Assembly =
            Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089")

        Dim pubTypesQuery = From type In asmbly.GetTypes()
                            Where type.IsPublic
                            From method In type.GetMethods()
                            Where method.ReturnType.IsArray = True
                            Let name = method.ToString()
                            Let typeName = type.ToString()
                            Group name By typeName Into methodNames = Group


        Console.WriteLine("Getting ready to iterate")
        For Each item In pubTypesQuery
            Console.WriteLine(item.methodNames)

            For Each type In item.methodNames
                Console.WriteLine(" " & type)
            Next
        Next
        Console.ReadKey()
    End Sub

End Module
```

The example uses the GetTypes method to return an array of types in the specified assembly. The Where Clause (Visual Basic) filter is applied so that only public types are returned. For each public type, a subquery is generated by using the

MethodInfo array that is returned from the GetMethods call. These results are filtered to return only those methods whose return type is an array or else a type that implements IEnumerable(Of T). Finally, these results are grouped by using the type name as a key.

Compiling the Code

Create a project that targets the .NET Framework version 3.5 or higher with a reference to System.Core.dll and a **Imports** statement for the System.Linq namespace.

See Also

LINQ to Objects (Visual Basic)

How to: Query an ArrayList with LINQ (Visual Basic)

Visual Studio 2015

When using LINQ to query non-generic IEnumerable collections such as ArrayList, you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. For example, if you have an ArrayList of `Student` objects, your From Clause (Visual Basic) should look like this:

```
Dim query = From student As Student In arrList
...
```

By specifying the type of the range variable, you are casting each item in the ArrayList to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the Cast(Of TResult) method. Cast(Of TResult) throws an exception if the specified cast cannot be performed. Cast(Of TResult) and OfType(Of TResult) are the two Standard Query Operator methods that operate on non-generic IEnumerable types. In Visual Basic, you must explicitly call the Cast(Of TResult) method on the data source to ensure a specific range variable type. For more information, seeType Relationships in Query Operations (Visual Basic).

Example

The following example shows a simple query over an ArrayList. Note that this example uses object initializers when the code calls the Add method, but this is not a requirement.

VB

```
Imports System.Collections
Imports System.Linq

Module Module1

    Public Class Student
        Public Property FirstName As String
        Public Property LastName As String
        Public Property Scores As Integer()
    End Class

    Sub Main()

        Dim student1 As New Student With {.FirstName = "Svetlana",
                                          .LastName = "Omelchenko",
                                          .Scores = New Integer() {98, 92, 81, 60}}
```

```
        Dim student2 As New Student With {.FirstName = "Claire",
                                          .LastName = "O'Donnell",
                                          .Scores = New Integer() {75, 84, 91, 39}}
        Dim student3 As New Student With {.FirstName = "Cesar",
                                          .LastName = "Garcia",
                                          .Scores = New Integer() {97, 89, 85, 82}}
        Dim student4 As New Student With {.FirstName = "Sven",
                                          .LastName = "Mortensen",
                                          .Scores = New Integer() {88, 94, 65, 91}}

        Dim arrList As New ArrayList()
        arrList.Add(student1)
        arrList.Add(student2)
        arrList.Add(student3)
        arrList.Add(student4)

        ' Use an explicit type for non-generic collections
        Dim query = From student As Student In arrList
                    Where student.Scores(0) > 95
                    Select student

        For Each student As Student In query
            Console.WriteLine(student.LastName & ": " & student.Scores(0))
        Next
        ' Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub

End Module
' Output:
'   Omelchenko: 98
'   Garcia: 97
```

See Also

LINQ to Objects (Visual Basic)

How to: Add Custom Methods for LINQ Queries (Visual Basic)

Visual Studio 2015

You can extend the set of methods that you can use for LINQ queries by adding extension methods to the IEnumerable(Of T) interface. For example, in addition to the standard average or maximum operations, you can create a custom aggregate method to compute a single value from a sequence of values. You can also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are Distinct(Of TSource), Skip(Of TSource), and Reverse(Of TSource).

When you extend the IEnumerable(Of T) interface, you can apply your custom methods to any enumerable collection. For more information, see Extension Methods (Visual Basic).

Adding an Aggregate Method

An aggregate method computes a single value from a set of values. LINQ provides several aggregate methods, including Average(Of TSource), Min(Of TSource), and Max(Of TSource). You can create your own aggregate method by adding an extension method to the IEnumerable(Of T) interface.

The following code example shows how to create an extension method called `Median` to compute a median for a sequence of numbers of type **double**.

VB

```
Imports System.Runtime.CompilerServices

Module LINQExtension

    ' Extension method for the IEnumerable(of T) interface.
    ' The method accepts only values of the Double type.
    <Extension()>
    Function Median(ByVal source As IEnumerable(Of Double)) As Double
        If source.Count = 0 Then
            Throw New InvalidOperationException("Cannot compute median for an empty
set.")
        End If

        Dim sortedSource = From number In source
                           Order By number

        Dim itemIndex = sortedSource.Count \ 2

        If sortedSource.Count Mod 2 = 0 Then
            ' Even number of items in list.
            Return (sortedSource(itemIndex) + sortedSource(itemIndex - 1)) / 2
        Else
```

```
            ' Odd number of items in list.
            Return sortedSource(itemIndex)
        End If
    End Function
End Module
```

You call this extension method for any enumerable collection in the same way you call other aggregate methods from the IEnumerable(Of T) interface.

Note

In Visual Basic, you can either use a method call or standard query syntax for the **Aggregate** or **Group By** clause. For more information, see Aggregate Clause (Visual Basic) and Group By Clause (Visual Basic).

The following code example shows how to use the `Median` method for an array of type **double**.

VB

```
Dim numbers1() As Double = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

Dim query1 = Aggregate num In numbers1 Into Median()

Console.WriteLine("Double: Median = " & query1)
```

VB

```
' This code produces the following output:
'
' Double: Median = 4.85
```

Overloading an Aggregate Method to Accept Various Types

You can overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that will take a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

To create an overload for each type

You can create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the **integer** type.

VB

```
' Integer overload

<Extension()>
Function Median(ByVal source As IEnumerable(Of Integer)) As Double
    Return Aggregate num In source Select CDbl(num) Into med = Median()
```

```
End Function
```

You can now call the `Median` overloads for both **integer** and **double** types, as shown in the following code:

VB

```
Dim numbers1() As Double = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

Dim query1 = Aggregate num In numbers1 Into Median()

Console.WriteLine("Double: Median = " & query1)
```

VB

```
Dim numbers2() As Integer = {1, 2, 3, 4, 5}

Dim query2 = Aggregate num In numbers2 Into Median()

Console.WriteLine("Integer: Median = " & query2)
```

VB

```
' This code produces the following output:
'
' Double: Median = 4.85
' Integer: Median = 3
```

To create a generic overload

You can also create an overload that accepts a sequence of generic objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the Func(Of T, TResult) delegate as a parameter. This delegate takes an object of generic type T and returns an object of type **double**.

VB

```
' Generic overload.

<Extension()>
Function Median(Of T)(ByVal source As IEnumerable(Of T),
                      ByVal selector As Func(Of T, Double)) As Double
    Return Aggregate num In source Select selector(num) Into med = Median()
End Function
```

You can now call the `Median` method for a sequence of objects of any type. If the type does not have its own method overload, you have to pass a delegate parameter. In Visual Basic, you can use a lambda expression for this purpose. Also, if you use the **Aggregate** or **Group By** clause instead of the method call, you can pass any value or

expression that is in the scope this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the Func(Of T, TResult) delegate parameter to the `Median` method for each case.

VB

```
Dim numbers3() As Integer = {1, 2, 3, 4, 5}

' You can use num as a parameter for the Median method
' so that the compiler will implicitly convert its value to double.
' If there is no implicit conversion, the compiler will
' display an error message.

Dim query3 = Aggregate num In numbers3 Into Median(num)

Console.WriteLine("Integer: Median = " & query3)

Dim numbers4() As String = {"one", "two", "three", "four", "five"}

' With the generic overload, you can also use numeric properties of objects.

Dim query4 = Aggregate str In numbers4 Into Median(str.Length)

Console.WriteLine("String: Median = " & query4)

' This code produces the following output:
'
' Integer: Median = 3
' String: Median = 4
```

Adding a Method That Returns a Collection

You can extend the IEnumerable(Of T) interface with a custom query method that returns a sequence of values. In this case, the method must return a collection of type IEnumerable(Of T). Such methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

VB

```
' Extension method for the IEnumerable(of T) interface.
' The method returns every other element of a sequence.

<Extension()>
Function AlternateElements(Of T)(
```

```
    ByVal source As IEnumerable(Of T)
    ) As IEnumerable(Of T)

    Dim list As New List(Of T)
    Dim i = 0
    For Each element In source
        If (i Mod 2 = 0) Then
            list.Add(element)
        End If
        i = i + 1
    Next
    Return list
End Function
```

You can call this extension method for any enumerable collection just as you would call other methods from the IEnumerable(Of T) interface, as shown in the following code:

VB

```
Dim strings() As String = {"a", "b", "c", "d", "e"}

Dim query = strings.AlternateElements()

For Each element In query
    Console.WriteLine(element)
Next

' This code produces the following output:
'
' a
' c
' e
```

See Also

IEnumerable(Of T)
Extension Methods (Visual Basic)