Visual Basic Features That Support LINQ

Visual Studio 2015

The name Language-Integrated Query (LINQ) refers to technology in Visual Basic that supports query syntax and other language constructs directly in the language. With LINQ, you do not have to learn a new language to query against an external data source. You can query against data in relational databases, XML stores, or objects by using Visual Basic. This integration of query capabilities into the language enables compile-time checking for syntax errors and type safety. This integration also ensures that you already know most of what you have to know to write rich, varied queries in Visual Basic.

The following sections describe the language constructs that support LINQ in enough detail to enable you to get started in reading the introductory documentation, code examples, and sample applications. You can also click the links to find more detailed explanations of how the language features come together to enable language-integrated query. A good place to start is Walkthrough: Writing Queries in Visual Basic.

Query Expressions

Query expressions in Visual Basic can be expressed in a declarative syntax similar to that of SQL or XQuery. At compile time, query syntax is converted into method calls to a LINQ provider's implementation of the standard query operator extension methods. Applications control which standard query operators are in scope by specifying the appropriate namespace with an **Imports** statement. Syntax for a Visual Basic query expression looks like this:

VB

```
Dim londonCusts = From cust In customers
                  Where cust.City = "London"
                  Order By cust.Name Ascending
                  Select cust.Name, cust.Phone
```

For more information, see Introduction to LINQ in Visual Basic.

Implicitly Typed Variables

Instead of explicitly specifying a type when you declare and initialize a variable, you can enable the compiler to infer and assign the type. This is referred to as *local type inference*.

Variables whose types are inferred are strongly typed, just like variables whose type you specify explicitly. Local type inference works only when you are defining a local variable inside a method body. For more information, see Option Infer Statement and Local Type Inference (Visual Basic).

The following example illustrates local type inference. To use this example, you must set **Option Infer** to **On**.

VB

```
' The variable aNumber will be typed as an integer.
Dim aNumber = 5
```

```
' The variable aName will be typed as a String.
Dim aName = "Virginia"
```

Local type inference also makes it possible to create anonymous types, which are described later in this section and are necessary for LINQ queries.

In the following LINQ example, type inference occurs if **Option Infer** is either **On** or **Off**. A compile-time error occurs if **Option Infer** is **Off** and **Option Strict** is **On**.

VB

```
' Query example.
' If numbers is a one-dimensional array of integers, num will be typed
' as an integer and numQuery will be typed as IEnumerable(Of Integer)--
' basically a collection of integers.

Dim numQuery = From num In numbers
               Where num Mod 2 = 0
               Select num
```

Object Initializers

Object initializers are used in query expressions when you have to create an anonymous type to hold the results of a query. They also can be used to initialize objects of named types outside of queries. By using an object initializer, you can initialize an object in a single line without explicitly calling a constructor. Assuming that you have a class named `Customer` that has public `Name` and `Phone` properties, along with other properties, an object initializer can be used in this manner:

VB

```
Dim aCust = New Customer With {.Name = "Mike",
                               .Phone = "555-0212"}
```

For more information, see Object Initializers: Named and Anonymous Types (Visual Basic).

Anonymous Types

Anonymous types provide a convenient way to temporarily group a set of properties into an element that you want to include in a query result. This enables you to choose any combination of available fields in the query, in any order, without defining a named data type for the element.

An *anonymous type* is constructed dynamically by the compiler. The name of the type is assigned by the compiler, and it might change with each new compilation. Therefore, the name cannot be used directly. Anonymous types are initialized in the following way:

VB

```
' Outside a query.
Dim product = New With {.Name = "paperclips", .Price = 1.29}

' Inside a query.
' You can use the existing member names of the selected fields, as was
' shown previously in the Query Expressions section of this topic.
Dim londonCusts1 = From cust In customers
                   Where cust.City = "London"
                   Select cust.Name, cust.Phone

' Or you can specify new names for the selected fields.
Dim londonCusts2 = From cust In customers
                   Where cust.City = "London"
                   Select CustomerName = cust.Name,
                   CustomerPhone = cust.Phone
```

For more information, see Anonymous Types (Visual Basic).

Extension Methods

Extension methods enable you to add methods to a data type or interface from outside the definition. This feature enables you to, in effect, add new methods to an existing type without actually modifying the type. The standard query operators are themselves a set of extension methods that provide LINQ query functionality for any type that implements IEnumerable(Of T). Other extensions to IEnumerable(Of T) include Count, Union, and Intersect.

The following extension method adds a print method to the String class.

VB

```
' Import System.Runtime.CompilerServices to use the Extension attribute.
<Extension()>
    Public Sub Print(ByVal str As String)
    Console.WriteLine(str)
End Sub
```

The method is called like an ordinary instance method of String:

VB

```
Dim greeting As String = "Hello"
greeting.Print()
```

For more information, see Extension Methods (Visual Basic).

Lambda Expressions

A lambda expression is a function without a name that calculates and returns a single value. Unlike named functions, a

lambda expression can be defined and executed at the same time. The following example displays 4.

VB

```
Console.WriteLine((Function(num As Integer) num + 1)(3))
```

You can assign the lambda expression definition to a variable name and then use the name to call the function. The following example also displays 4.

VB

```
Dim add1 = Function(num As Integer) num + 1
Console.WriteLine(add1(3))
```

In LINQ, lambda expressions underlie many of the standard query operators. The compiler creates lambda expressions to capture the calculations that are defined in fundamental query methods such as **Where**, **Select**, **Order By**, **Take While**, and others.

For example, the following code defines a query that returns all senior students from a list of students.

VB

```
Dim seniorsQuery = From stdnt In students
                   Where stdnt.Year = "Senior"
                   Select stdnt
```

The query definition is compiled into code that is similar to the following example, which uses two lambda expressions to specify the arguments for **Where** and **Select**.

VB

```
Dim seniorsQuery2 = students.
    Where(Function(st) st.Year = "Senior").
    Select(Function(s) s)
```

Either version can be run by using a **For Each** loop:

VB

```
For Each senior In seniorsQuery
    Console.WriteLine(senior.Last & ", " & senior.First)
Next
```

For more information, see Lambda Expressions (Visual Basic).

See Also

Language-Integrated Query (LINQ) (Visual Basic)

Getting Started with LINQ in Visual Basic
LINQ and Strings (Visual Basic)
Option Infer Statement
Option Strict Statement