# LINQ (Language-Integrated Query)

The => token is called the lambda operator. It is used in lambda expressions to separate the input variables on the left side from the lambda body on the right side. Lambda expressions are inline expressions similar to anonymous methods but more flexible; they are used extensively in LINQ queries that are expressed in method syntax. For more information, see [Lambda Expressions](https://msdn.microsoft.com/en-us/library/bb397687.aspx).

The following example shows two ways to find and display the length of the shortest string in an array of strings. The first part of the example applies a lambda expression (w => w.Length) to each element of the words array and then uses the [Min<TSource>](https://msdn.microsoft.com/en-us/library/bb548864.aspx) method to find the smallest length. For comparison, the second part of the example shows a longer solution that uses query syntax to do the same thing.

C#

string[] words = { "cherry", "apple", "blueberry" };

// Use method syntax to apply a lambda expression to each element

// of the words array.

int shortestWordLength = words.Min(w => w.Length);

Console.WriteLine(shortestWordLength);

// Compare the following code that uses query syntax.

// Get the lengths of each word in the words array.

var query = from w in words

select w.Length;

// Apply the Min method to execute the query and get the shortest length.

int shortestWordLength2 = query.Min();

Console.WriteLine(shortestWordLength2);

A [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e(v=vs.80).aspx) is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate.

A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the lambda operator =>, which is read as “goes to”. The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block.

The lambda expression x => x \* x is read “x goes to x times x.”

## What are Delegates?

If I can put it into simple words, Delegate is a pointer to a method. Delegate can be passed as a parameter to a method. We can change the method implementation dynamically at run-time

Delegates can be executed synchronous or asynchronously. Code sample mentioned above is an example of synchronous processing. To make it an Asynchronous process, we need to use BeginInvoke method.

Hide   Copy Code

funcPointer.BeginInvoke("Hello", 1, null, null);

The difference between **Func** and **Action** is simply whether you want the delegate to return a value (use Func) or not (use Action).

Func is probably most commonly used in LINQ - for example in projections:

list.Select(x => x.SomeProperty)

or filtering:

list.Where(x => x.SomeValue == someOtherValue)

or key selection:

list.Join(otherList, x => x.FirstKey, y => y.SecondKey, ...)

Action is more commonly used for things like List<T>.ForEach: execute the given action for each item in the list. I use this less often than **Func**, although I *do* sometimes use the parameterless version for things like Control.BeginInvoke and Dispatcher.BeginInvoke.

Predicate is just a special cased Func<T, bool> really, introduced before all of the Func and most of the Action delegates came along. I suspect that if we'd already had Func and Action in their various guises, **Predicate** wouldn't have been introduced... although it *does* impart a certain meaning to the use of the delegate, whereas **Func** and **Action** are used for widely disparate purposes.

**Predicate** is mostly used in List<T> for methods like FindAll and RemoveAll.

### **Action<T>** This delegate was introduced in Framework 2.0. We can now use Action<in T> delegate to pass a method as a parameter without explicitly defining the delegate. The compiler will take care of it. This delegate can accept parameters but without a return type.

### define the delegate

### **Func< T, TResult>** This was introduced in Framework3.5. This delegate is different from Action<T> in the sense that it supports for return value. return value To call this delegate we can use Console.WriteLine(“Using Func<> :” + fn(6, 6));

### The Predicate will always return a boolean, by definition. Predicate<T> is basically identical to Func<T,bool> . Predicates are very useful in programming. They are often used to allow you to provide logic at runtime, that can be as simple or as complicated as necessary.

### LINQ interview questions

**what are Lambda expressions?**   
  
A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.   
All lambda expressions use the lambda operator =>, which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block.   
  
The => operator has the same precedence as assignment (=) and is right-associative.   
  
Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as Where and Where(IQueryable, String, array[]).

**How do you assign a Lambda expression to a delegate?**

delegate int del(int i);   
del myDelegate = x => x \* x;   
int j = myDelegate(5); //j = 25

**Can we write a lamba expression on the left side of the is or as operator?**

Lambdas are not allowed on the left side of the is or as operator.

**What is Expression Lambda?**

A lambda expression with an expression on the right side is called an expression lambda. Expression lambdas are used extensively in the construction of Expression Trees. An expression lambda returns the result of the expression and takes the following basic form:

(input parameters) => expression

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

(x, y) => x == y

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

(int x, string s) => s.Length > x

Specify zero input parameters with empty parentheses:

() => SomeMethod()

Note in the previous example that the body of an expression lambda can consist of a method call. However, if you are creating expression trees that will be consumed in another domain, such as SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

**What is Statement Lambda?**

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

(input parameters) => {statement;}

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

delegate void TestDelegate(string s);   
…   
TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };   
myDel("Hello");

**What is the difference between Statement Lambdas and Expression Lambdas ?**

Expression lambdas are used extensively in the construction of Expression Trees.

Statement lambdas cannot be used to create expression trees.

**What is Type Inference in Lambdas ?**

When writing lambdas, you often do not have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the underlying delegate type, and other factors as described in the C# 3.0 Language Specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. So if you are querying an IEnumerable, then the input variable is inferred to be a Customer object, which means you have access to its methods and properties:

customers.Where(c => c.City == "London");

**Explain Rules of lambdas?**   
The lambda must contain the same number of parameters as the delegate type.   
Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.   
The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

**Does lambda expressions in themselves do have a type?**

lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or Expression type to which the lambda expression is converted.

**Explain about Variable Scope in Lambda Expressions?**

Lambdas can refer to outer variables that are in scope in the enclosing method or type in which the lambda is defined. Variables that are captured in this manner are stored for use in the lambda expression even if variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression.

**List out the rules apply to variable scope in lambda expressions?**

A variable that is captured will not be garbage-collected until the delegate that references it goes out of scope.   
Variables introduced within a lambda expression are not visible in the outer method.   
A lambda expression cannot directly capture a ref or out parameter from an enclosing method.   
A return statement in a lambda expression does not cause the enclosing method to return.   
A lambda expression cannot contain a goto statement, break statement, or continue statement whose target is outside the body or in the body of a contained anonymous function.

**What are Expression Trees?**

Expression trees represent language-level code in the form of data. The data is stored in a tree-shaped structure. Each node in the expression tree represents an expression, for example a method call or a binary operation such as x <>

// Create an expression tree.

Expression<func> exprTree = num => num <>

// Decompose the expression tree.ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];BinaryExpression operation = (BinaryExpression)exprTree.Body;ParameterExpression left = (ParameterExpression)operation.Left;ConstantExpression right = (ConstantExpression)operation.Right;

**are Expression Trees Immutable?**

Expression trees are immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and modifying it. You can use an expression tree visitor to traverse the existing expression tree.

**What is LINQ?**

Language-Integrated Query (LINQ) is a set of features 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.

**Why LINQ is required?**

Language-Integrated Query (LINQ) 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.

Linq vs Lambda

This is LINQ (using query syntax):

var \_Results = from item in \_List

where item.Value == 1

select item;

This is also LINQ (using method syntax):

var \_Results = \_List.Where(x => x.Value == 1);

It's interesting to note that [both of these flavors](http://msdn.microsoft.com/en-us/library/bb397947.aspx) will end up producing the exact same code. The compiler offers you a service by allowing you to express your wishes in the manner that you prefer.

And **this** is a lambda:

x => x.Value == 1

When you choose to use method syntax, LINQ is almost always seen around lambda expressions. But [LINQ](http://msdn.microsoft.com/en-us/library/bb397926.aspx) and [lambdas](http://msdn.microsoft.com/en-us/library/bb397687.aspx) are two totally different things, both of which can be used by themselves.

**Update:** As svick rightly points out, LINQ with query syntax is *also* implemented using lambda expressions (as mentioned earlier, the compiler allows you to write in query syntax but effectively transforms it to method syntax behind your back). This is just piling on the fact that both flavors are totally equivalent and will behave the same way (e.g. lambda expressions may cause [closures](http://csharpindepth.com/Articles/Chapter5/Closures.aspx) to be created).

This post covers one of the most important and frequently misunderstood LINQ features. Understanding deferred execution is a rite of passage that LINQ developers must undergo before they can hope to harness the full power of this technology. The contents of this post assumes an intermediate understanding of LINQ.

**Note**: *A video that accompanies this post can be found*[*here*](http://blogs.msdn.com/charlie/archive/2007/12/13/deferred-execution-video.aspx). *The sample code from post is found in the attached program called****DeferredExecution*.**

**Note:***In several places in this explanation of deferred execution I will refer to a data structure called an expression tree. To understand this portion of the text, you need only grasp that an expression tree is a data structure like a list or queue. It holds a LINQ to SQL query — not the results of the query, but the actual elements of the query itself. In some future post I will cover expression trees in more depth.*

Consider this simple LINQ to SQL query:

var query = from customer in db.Customers << Query does

where customer.City == “Paris” << ***not*** execute

select customer; << here

It is easy to assume that these few lines of code execute a query against a database. In fact, they do not. This code merely captures the idea of the query in a data structure called an expression tree. The tree contains information about the table you want to query, the question you want to ask of the table, and the result you want to return. It does not, however, actually execute the query!

In LINQ, execution of a query is usually deferred until the moment when you actually request the data. For instance, the following code would cause the query to execute:

foreach (var Customer in query) << Query executes here

{

Console.WriteLine(Customer.CompanyName);

}

Query expressions often do not cause code to be executed. They only define the question you want to ask. If this were not the case, then it would be difficult or impossible for LINQ to break queries down into a relational algebra which makes composability possible, and which allows developers who care about such things to optimize their code. Deferred execution makes it possible for you to compose quite complex queries from various components without expending the clock cycles necessary to actually query the data. Or at least the data will not be queried until it is absolutely necessary.

Let’s make a slight change to the query shown above:

var query = (from customer in db.Customers << Query executes here

where customer.City == “Paris”

select customer).Count();

This time the query will be run when the execution point moves past it. The code executes when you call **Count()**, as it would when you call any of the other operators that must iterate over the result of a query in order to return a value that is not **IEnumerable<T>** or **IQueryable<T>**. The **ToList**() operator the query code to execute immediately because it returns a **List<T>** instead of **IQueryable<T>**. Consider the following code:

public void SimpleQuery01()

{

db.Log = Console.Out;

// Query executes here because it returns a List<T>

List<Customer> list = (from customer in db.Customers

where customer.City == “Paris”

select customer).ToList();

foreach (var Customer in list)

{

Console.WriteLine(Customer.CompanyName);

}

}

One of the last things the LINQ to SQL provider does before executing a query is to create the SQL it will send across the wire. This fact gives us a clue we can use when trying to determine the exact moment when a query executes.

Assigning **db**.**Log** to **Console**.**Out** as we do in the first line of this query ensures that the SQL for our query will be written to the console as soon as it is generated. When the code shown above is run, you will see the log written to the screen as the query expression executes.

Consider the following code, which does not call **ToList()**:

db.Log = Console.Out;

var query = from customer in db.Customers

where customer.City == “Paris”

select customer;

foreach (var Customer in query) << Query Executes here

{

Console.WriteLine(Customer.CompanyName);

}

If you step through this code with the debugger while watching the console window, you will see that the SQL is not generated until the **foreach** statement executes. When we call **ToList**(), the SQL is written to the screen earlier, providing the evidence that the query itself executes earlier:

List<Customer> list = (from customer in db.Customers

where customer.City == “Paris”

select customer).ToList(); << Query Executes here

Let’s look at deferred execution from a slightly different angle:

public void SimpleQuery03()

{

string city = “London”;

var query = from c in db.Customers

where c.City == city

select new { c.City };

city = “Madrid”;

foreach (var q in query) << Query Executes here

{

Console.WriteLine(q);

}

}

This method will print **Madrid** to the screen rather than **London**. By changing the value of the variable **city** to **Madrid** just before the **foreach** statement we ensure that **Madrid**, rather than **London** is included in the SQL that is sent to the server. The point being that the execution of the query expression merely generates an expression tree, it does not send SQL to the database.

Let’s take a look at one final example, just to drive this very important point home. Consider this code:

public void DeleteInsert01()

{

db.Log = Console.Out;

string customerId = “WIDGE”;

// Query expression to be used repeatedly

var query = from c in db.Customers

where c.CustomerID == customerId

select c;

Console.WriteLine(“Count before insert: {0}”, query.Count()); << SQL Sent

// Object Initializer

var newCustomer = new Customer

{

CustomerID = customerId,

CompanyName = “Microsoft”,

ContactName = “John Doe”,

ContactTitle = “Sales Manager”,

Address = “1 Microsoft Way”,

City = “Redmond”,

Region = “WA”,

PostalCode = “98052”,

Country = “USA”,

Phone = “(425) 555-1234”,

Fax = null

};

// Insert

db.Customers.InsertOnSubmit(newCustomer);

db.SubmitChanges();

Console.WriteLine(“Count after insert: {0}”, query.Count()); << SQL Sent

// Delete

db.Customers.DeleteAllOnSubmit(query);

db.SubmitChanges();

Console.WriteLine(“Count after delete: {0}”, query.Count()); << SQL Sent

}

This first call to **WriteLine**() shows the initial number of records with a **CustomerId** of **WIDGE**. The value returned is zero. The second **WriteLine** shows the value at 1, because a record with that value has been inserted. The final call shows the count as back at zero, because the inserted record was deleted

Notice that query expression used to retrieve the count from the server is written only once near top of the method. At each of the three **WriteLine** statements this query is composed with the **Count()** operator and executed. The following SQL was sent to the server three times:

SELECT COUNT(\*) AS [value]   
FROM [dbo].[Customers] AS [t0]   
WHERE [t0].[CustomerID] = @p0

SQL of a different sort was also sent to the server when **DeleteAllOnSubmit** was called.

Suppose we rewrote the method the method to look like this:

public void DeleteInsert02()

{

db.Log = Console.Out;

string customerId = “WIDGE”;

// Execution of the query occurs here

var query = (from c in db.Customers << Query executes

where c.CustomerID == customerId

select c).Count();

Console.WriteLine(“Count before insert: {0}”, query); << No execution

// Object Initializer

var newCustomer = new Customer

{

CustomerID = customerId,

CompanyName = “Microsoft”,

ContactName = “John Doe”,

ContactTitle = “Sales Manager”,

Address = “1 Microsoft Way”,

City = “Redmond”,

Region = “WA”,

PostalCode = “98052”,

Country = “USA”,

Phone = “(425) 555-1234”,

Fax = null

};

// Insert

db.Customers.InsertOnSubmit(newCustomer);

db.SubmitChanges();

Console.WriteLine(“Count after insert: {0}”, query); << No execution

var deleteQuery = from c in db.Customers

where c.CustomerID == customerId

select c;

// Delete

db.Customers.DeleteAllOnSubmit(deleteQuery);

db.SubmitChanges();

Console.WriteLine(“Count after delete: {0}”, query); << No execution

}

This time we get the count of records in the database when the query expression executes near the start of the method. The correct value, which is zero, is returned from the database at that time and is sent to the console when the first **WriteLine** statement executes. The second two times that **WriteLine** is called, the same value of zero that was returned from the initial query is sent to the server. As a result, invalid data is reported to the user! This illustrates why it is so important that you understand deferred execution.

Deferred execution is a two edged sword:

* Composable queries and deferred execution work together to make LINQ an unusual rich query language. If you properly understand these features of LINQ you will be able to write less code, that executes faster, in order to accomplish more.
* As you have seen, if we don’t properly understand deferred execution then we can end up with a sheepish grin on our face. Queries might not execute when we expect them to, and we might report erroneous results to the user.

**Note**: *Deferred execution applies to all varieties of LINQ, including LINQ to SQL, LINQ to Objects and LINQ to XML. However, of the three, it is only LINQ to SQL that returns an expression tree by default. Or more specifically, it returns an instance of the IQueryable interface that references an expression tree. A query that returns****IEnumerable****still supports deferred execution, but at least some larger portion of the result is likely to have been generated than is the case with LINQ to SQL. In other words, all types of LINQ support deferred execution, but LINQ to SQL supports it more fully than LINQ to Objects or LINQ to XML.*

What is a Lambda expression?  
  
  
A Lambda expression is nothing but an Anonymous Function, can contain expressions and statements. Lambda expressions can be used mostly to create delegates or expression tree types. Lambda expression uses lambda operator => and read as 'goes to' operator.   
  
Left side of this operator specifies the input parameters and contains the expression or statement block at the right side.   
1. What is Language Integrated Query (LINQ)?  
LINQ is a programming model that is the composition of general-purpose standard query operators that allow you to work with data, regardless of the data source in any .NET based programming language. It is the name given to a set of technologies based on the integration of query capabilities into any .NET language.  
2. What are LINQ query expressions?  
A LINQ query, also known as a query expression, consists of a combination of query clauses that identify the data sources for the query. It includes instructions for sorting, filtering, grouping, or joining to apply to the source data. The LINQ query expressions syntax is similar to the SQL syntax. It specifies what information should be retrieved from the data source.  
3. Write the basic steps to execute a LINQ query.  
The following are the three basic steps to execute a LINQ query:  
• Obtain the data source (The data source can be either an SQL database or an XML file)  
• Create a query  
• Execute the query  
4. Write the basic syntax of a LINQ query in Visual Basic as well as in C#.  
In Visual Basic, the basic syntax of a LINQ query starts with the From clause and ends with the Select or Group By clause. In addition, you can use the Where, Order By, and Order By Descending clauses to perform additional functions, such as filtering data and generating the data in a specific order.  
  
In C#, the basic syntax of a LINQ query starts with the From clause and ends with the Select or group by clause. In addition, you can use the where, orderby, and Orderby descending clauses to perform additional functions, such as filtering data and generating the data in a specific order.  
5. In which statement the LINQ query is executed?  
A LINQ query is executed in the For Each statement in Visual Basic and in the foreach statement in C#.  
6. In LINQ, lambda expressions underlie many of the standard query operators. Is it True or False?  
7. What is PLINQ?  
PLINQ stands for Parallel Language Integrated Query. It is the parallel implementation of LINQ, in which a query can be executed by using multiple processors. PLINQ ensures the scalability of software on parallel processors in the execution environment. It is used where data grows rapidly, such as in telecom industry or where data is heterogeneous.   
  
PLINQ also supports all the operators of LINQ. In addition, you can query 'collections by using PLINQ. It can also run several LINQ queries simultaneously and makes use of the processors on the system. Apart from this, PLINQ uses parallel execution, which helps in running the queries quickly. Parallel execution provides a major performance improvement to PLINQ over certain types of legacy code, which takes too much time to execute.  
8. What are the different Visual Basic features that support LINQ?  
Visual Basic includes the following features that support LINQ:  
• Anonymous types - Enables you to create a new type based on a query result.  
• Implicitly typed variables - Enables the compiler to infer and assign a type when you declare and initialize a variable.  
• Extension method - Enables you to extend an existing type with your own methods without modifying the type itself.  
9. What is the function of the DISTINCT clause in a LINQ query?  
The DISTINCT clause returns the result set without the duplicate values.  
10. What is the DataContext class and how is it related to LINQ?  
After you add a LINQ to SQL Classes item to a project and open the O/R Designer, the empty design surface represents an empty DataContext class ready to be configured. The DataContext class is a LINQ to SQL class that acts as a conduit between a SQL Server database and the LINQ to SQL entity classes mapped to that database. This class contains the connection string information and the methods for connecting to a database and manipulating the data in the database. It is configured with connection information provided by the first item that is dragged onto the design surface.  
11. What is the difference between the Take and Skip clauses?  
The Take clause returns a specified number of elements. For example, you can use the Take clause to return two values from an array of numbers. The Skip clause skips the specified number of elements in the query and returns the rest. For example, you can use the Skip clause to skip the first four strings in an array of strings and return the remaining array of string.  
12. What is Object Relational Designer (0/R Designer)?  
The 0/R Designer provides a visual design surface to create LINQ to SQL entity classes and associations (relationships) that are based on objects in a database.  
13. Which interface implements the standard query operators in LINQ?  
The standard query operators implement the IEnumerable<t> or the IQueryable<t> interface in C# and theIEnumerable(Of T) or the IQueryable(Of T) interface in Visual Basic.  
14. What are standard query operators in LINQ?  
The standard query operators in LINQ are the extension methods that form the LINQ pattern. These operators form an API that enables querying of any .NET array or collection. It operates on sequences and allows you to perform operations, such as determining if a value exists in the sequence and performing an aggregated function, such as a summation over a sequence.  
15. On what parameter does the GroupBy clause group the data?  
The GroupBy clause groups the elements that share a common attribute.  
16. What is a LinqDataSource control?  
The LinqDataSource control enables you to use LINQ. in an ASP.NET Web page by setting the properties in the markup text. You can use the control retrieve or modify data. It is similar to the SqIDataSource andObjectDataSource controls in the sense that it can be used to declaratively bind other ASP.NET controls on a page to a data source. The difference is that instead of binding directly to a database or to a generic class, theLinqDataSource control is designed to bind a LINQ enabled data model.  
17. How can you open the O/R Designer?  
You can open the O/R Designer by adding a new LINQ to SQL Classes item to a project.  
18. The standard query operators are themselves a set of extension methods that provide the LINQ query functionality for any type that implements the IEnumerable<t> interface in Visual Basic. Is it True or False?  
19. What are lambda expressions in LINQ?  
A lambda expression is a function without a name that calculates and returns a single value. All lambda expressions use the lambda operator =>, which read as goes to. The left side of the lambda operator specifies the input parameters and the right side holds the expression or statement block.  
20. Before you query a DataSet object by using LINQ to DataSet, you must first populate the dataset How can you do this?  
You can load the data into the dataset by using different methods, such as:  
• Using the DataAdapter class  
• Using LINQ to SQL  
21. What are the different implementations of LINQ?  
The different implementations of LINQ are:  
• LINQ to SQL - Refers to a component of.NET Framework version 3.5 that provides a run-time infrastructure to manage relational data as objects.  
• LINQ to DataSet - Refers to a component that makes it easier and faster to query over data cached in a DataSet object.  
• LINQ to XML - Provides an in-memory XML programming interface.  
• LINQ to Objects - Refers to the use of LINQ queries with any IEnumerable or IEnumerable(T)collection directly, without the use of an intermediate LINQ provider or API, such as LINQ to SQL or LINQ to XML.  
22. Which command-line tool generates code and mapping for the LINQ to SQL component of .NET Framework?  
The SqlMetal.exe command-line tool generates code and map the LINQ to SQL component.  
23. Name the control that exposes the LINQ features to Web developers through the ASP.NET data-source control architecture.  
The LinqDataSource control exposes the LINQ features to Web developers through the ASP.NET data-source control architecture.  
24. What is the difference between the Select clause and SelectMany() method in LINQ?  
Both the Select clause and SelectMany() method are used to produce a result value from a source of values. The difference lies in the result set. The Select clause is used to produce one result value for every source value. The result value is a collection that has the same number of elements from the query. In contrast, theSelectMany() method produces a single result that contains a concatenated collection from the query.

**There are 2 ways to write LINQ queries using these Standard Query Operators**  
**1.** **Using Lambda Expressions.** We discussed Lambda Expressions in detail in [Part 99](http://www.youtube.com/watch?v=LDgQ-spnrYY&list=PLAC325451207E3105&index=100) of [C# Tutorial](http://www.youtube.com/playlist?list=PLAC325451207E3105)  
  
**2.** **Using SQL like query expressions**  
  
The **Standard Query Operators**are implemented as extension methods on IEnumerable<T> interface. We will discuss, what extension methods are and how to implement them in a later video session.  
  
**For now let's focus on the 2 ways of writing a LINQ query**. From a performance perspective there is no difference between the two. Which one to use depends on your personal preference. But keep in mind, behind the scene, LINQ queries written using SQL like query expressions are translated into their lambda expressions before they are compiled.

**What are Extension Methods**  
**According to MSDN**, Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.   
**Extension methods are a special kind of static method**, but they are called as if they were instance methods on the extended type.   
  
**For client code written in C# and Visual Basic**, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

**LINQ Standard Query Operators**also called as **LINQ extension methods**can be broadly classified into the following categories  
Aggregate Operators  
Grouping Operators  
Restriction Operators  
Projection Operators  
Set Operators  
Partitioning Operators  
Conversion Operators  
Element Operators  
Ordering Operators  
Generation Operators  
Query Execution  
Join Operators  
Custom Sequence Operators  
Quantifiers Operators  
Miscellaneous Operators

What is a Predicate?

A predicate is a function to test each element for a condition

In the following example, the Lambda expression(**num => num % 2 == 0**) runs for each element in List<int>.If the number is divisible by 2, then a boolean value true is returned otherwise false.

The following **5 standard LINQ query operators belong to Ordering Operators** category  
OrderBy  
OrderByDescending  
ThenBy  
ThenByDescending  
Reverse

### Partitioning Operators

**Take**method returns a specified number of elements from the start of the collection. The number of items to return is specified using the count parameter this method expects.  
  
**Skip**method skips a specified number of elements in a collection and then returns the remaining elements. The number of items to skip is specified using the count parameter this method expects.   
  
**Please Note:** For the same argument value, the Skip method returns all of the items that the Take method would not return.  
  
**TakeWhile**method returns elements from a collection as long as the given condition specified by the predicate is true.   
  
**SkipWhile**method skips elements in a collection as long as the given condition specified by the predicate is true, and then returns the remaining elements.

**deferred execution**. LINQ queries have two different behaviors of execution  
**1.** Deferred execution  
**2.** Immediate execution   
  
**LINQ operators can be broadly classified into 2 categories based on the behaviour of query execution**  
**1. Deferred or Lazy Operators -**These query operators use deferred execution.   
Examples - select, where, Take, Skip etc

**2. Immediate or Greedy Operators -**These query operators use immediate execution.   
Examples - count, average, min, max, ToList etc………………………………………….

# DEFERRED EXECUTION VS IMMEDIATE EXECUTION IN LINQ

LINQ provides two different behaviors of Query Execution –

* **Deferred Execution**
* **Immediate Execution**

|  |  |
| --- | --- |
| **Deferred/Lazy Operators** | **Immediate/Greedy Operators** |
| Query is not executed at the point of its declaration. | Query is executed at the point of its declaration. |
| Projection Operator – Select, SelectMany Restriction Operator – WherePaging Operator – Take, Skip | Aggregate Functions – Count, Average, Min, Max, Sum Element Operators – First, Last, SingleToList, ToArray, ToDictionary |

**Deferred Execution**

By default, LINQ uses deferred execution.

When we write a LINQ query, it doesn’t execute by itself. It executes, when we access the query results.In other words, execution of the query is deferred until the query variable is iterated over in a foreach loop.

Benefits of Deferred Execution –

• It avoids unnecessary query execution and hence improves performance.

• Query construction and Query execution are decoupled, so we can create the LINQ query in several steps.

• A deferred execution query is reevaluated when you re-enumerate – hence we always get the latest data.

Deferred Execution Example

List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            // LINQ Query is only defined here and is not executed at this point

            // If the query is executed at this point, the result should not display Tim

            IEnumerable<Student> result = from student in listStudents

                                          where student.TotalMarks == 800

                                          select student;

            // Add a new student object with TotalMarks = 800 to the source

            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is actually executed when we iterate thru the sequence

            // using the foreach loop. This is proved as Tim is also included in the result

            foreach (Student s in result)

            {

                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

            }

This is because the execution of the query was deferred until the query variable was iterated using a foreach loop.

**Immediate Execution**

Immediate execution is the reverse of deferred execution. It forces the LINQ query to execute and gets the result immediately.

To force immediate execution of a query, that return a singleton value, we can use the aggregate operators or element operators.

To force immediate execution of a query that does not produce a singleton value, we can call the ToList method, the ToDictionary method, or the ToArray method on a query or query variable.

Immediate Execution Example

 public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            // Since we are using ToList() which is a greedy operator

            // the LINQ Query is executed immediately at this point

            IEnumerable<Student> result = (from student in listStudents

                                           where student.TotalMarks == 800

                                           select student).ToList();

            // Adding a new student object with TotalMarks = 800 to the source

            // will have no effect on the result as the query is already executed

            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is executed at the point where it is defined.

            // This is proved as Tim is not included in the result

            foreach (Student s in result)

            {

                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

            }

        }

    }

This is because the execution of the query was done immediately, when it was constructed – since we used the ‘ToList()’ operator.

**The following standard LINQ query operators belong to Conversion Operators category**ToList  
ToArray  
ToDictionary  
ToLookup  
Cast  
OfType  
AsEnumerable   
AsQueryable   
  
   
  
**ToList operator** extracts all of the items from the source sequence and returns a new **List<T>**. This operator causes the query to be executed immediately. This operator does not use deferred execution.

**ToArray** operator extracts all of the items from the source sequence and returns a new Array. This operator causes the query to be executed immediately. This operator does not use deferred execution.

**ToDictionary** operator extracts all of the items from the source sequence and returns a new Dictionary. This operator causes the query to be executed immediately. This operator does not use deferred execution.

**ToLookup** creates a Lookup. Just like a dictionary, a Lookup is a collection of key/value pairs. A dictionary cannot contain keys with identical values, where as a Lookup can.

**AsEnumerable**

**AsQueryable**operators in LINQ. Both of these operators belong to **Conversion Operators**category.

**AsQueryable operator:** There are 2 overloaded versions of this method.   
  
One overloaded version converts System.Collections.IEnumerable to System.Linq.IQueryable  
  
The other overloaded version converts a generic System.Collections.Generic.IEnumerable<T> to a generic System.Linq.IQueryable<T>  
  
The main use of **AsQueryable**operator is unit testing to mock a **queryable**data source using an in-memory data source. We will discuss this operator in detail with examples in unit testing video series.