LINQ

Chapter – 1 (Introducing LINQ)

Language-Integrated Query (LINQ) is an innovation introduced in the .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data. 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#. You write queries against strongly typed collections of objects by using language keywords and familiar operators.

LINQ (Language Integrated Query) is uniform query syntax in C# and VB.NET used to save and retrieve data from different sources. It is integrated in C# or VB, thereby eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources. For example, SQL is a Structured Query Language used to save and retrieve data from a database. In the same way, LINQ is a structured query syntax built in C# and VB.NET used to save and retrieve data from different types of data sources like an Object Collection, SQL server database, XML, web service etc. Hence You can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface.

LINQ was introduced in .NET version 3.5 to allow a programmer to query data from many kinds of data sources without knowing any external language. Querying is the process of obtaining data from a data source. LINQ makes it very easy for you to query data from different kinds of data sources. LINQ is integrated to both C# and VB, and multiple special keywords and syntax for querying using LINQ have been added.
Before the arrival of LINQ, programmers write a different set of codes for querying different data sources. For example, they have to write codes for querying an SQL database using an SQL command or using XPath for querying XML files. With LINQ now in the programmer's arsenal, querying different data sources requires only the knowledge of the LINQ keywords and methods that were added in .NET 3.5.

There are multiple flavors of LINQ. This is made possible by LINQ providers as seen in Figure 1. Visual Studio already includes some of this provider such as LINQ to Objects. This section of the site will focus on LINQ to Objects which is used to query a collection of objects in your code that implements the IEnumerable<T> interface. Examples of such objects are arrays and lists or a custom collection that you created. There is also an LINQ to SQL which is specifically designed to make it easier to query SQL Server databases. For querying XML files, you can use the LINQ to XML. You can extend LINQ to query more kinds of data sources. You can create you own providers if you want to support querying another type of data source using LINQ. The querying techniques that will be thought in the following lessons can be applied on the different flavors of LINQ.

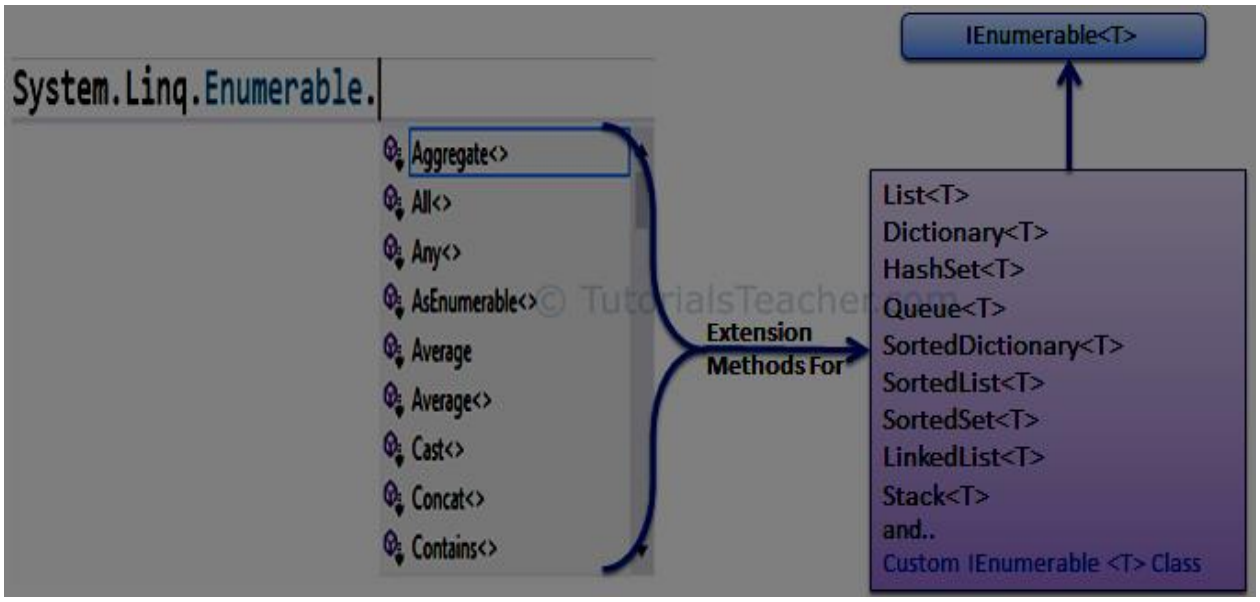
Understanding the LINQ :

LINQ is nothing but the collection of extension methods. System.Linq namespace includes the necessary classes & interfaces for LINQ. Enumerable and Queryable are two main static classes of LINQ API that contain extension methods for LINQ.

Enumerable:

Enumerable class includes extension methods for IEnumerable<T>interface. Basically all the collection types which is included in System.Collections.Generic namespaces e.g. List<T>, Dictionary<T>, SortedList<T>, Queue<T>, HashSet<T>, LinkedList<T> etc implement IEnumerable<T> interface.

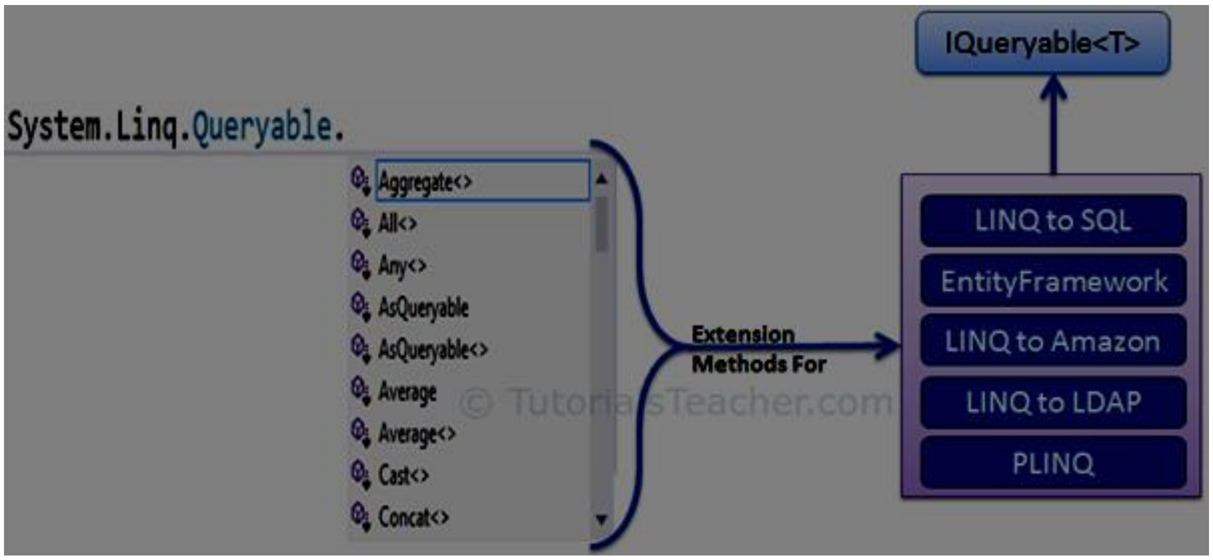
The following figure illustrates that the extension methods included in Enumerable class can be used with generic collection in C# or VB.Net.

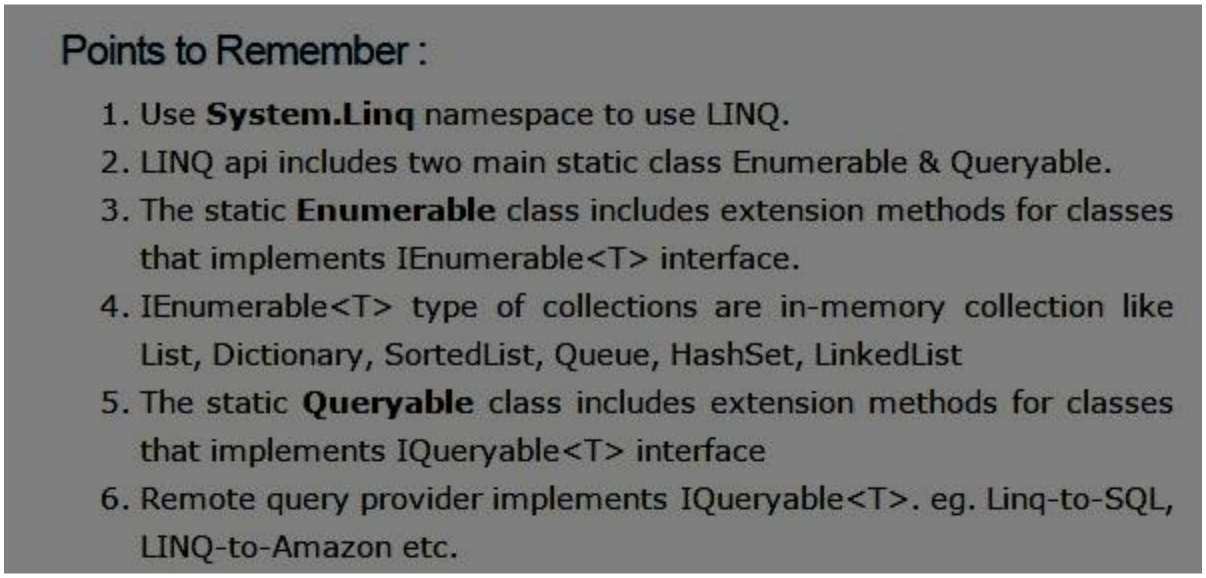


Queryable:

The Queryable class includes extension methods for IQueryable<t>interface. IQueryable<T> is used to provide querying capabilities against a specific data source where the type of the data is known. For example, Entity Framework api implements IQueryable<T> interface to support LINQ queries with underlying database like SQL Server.

Also, there are APIs available to access third party data; for example, LINQ to Amazon provides the ability to use LINQ with Amazon web services to search for books and other items by implementing IQueryable interface.
The following figure illustrates that the extension methods included in Queryable class can be used with various native or third party data providers.





Points to Remember :

1. Use **System.Linq** namespace to use LINQ.
2. LINQ api includes two main static class Enumerable & Queryable.
3. The static **Enumerable** class includes extension methods for classes that implements IEnumerable<T> interface.
4. IEnumerable<T> type of collections are in-memory collection like List, Dictionary, SortedList, Queue, HashSet, LinkedList
5. The static **Queryable** class includes extension methods for classes that implements IQueryable<T> interface
6. Remote query provider implements IQueryable<T>. eg. Linq-to-SQL, LINQ-to-Amazon etc.

Chapter – 2 (Syntax for LINQ)

Before starting with the syntax with LINQ , you have to understand the three important concepts in C# :

A. Extension Methods.
B. Delegates.
C. Anonymous Methods.
D. Lambda Expression and Statements.

A. Extension Methods:

Extension methods introduced in C# 3.0 enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

Features of Extension Methods :

1. It is a static method.
2. It must be located in a static class.
3. It uses the "this" keyword as the first parameter with a type in .NET and this method will be called by a given type instance on the client side.
4. It also shown by VS intellisense. When we press the dot (.) after a type instance, then it comes in VS intellisense.
5. An extension method should be in the same namespace as it is used or you need to import the namespace of the class by a using statement.
6. You can give any name for the class that has an extension method but the class should be static.
7. If you want to add new methods to a type and you don't have the source code for it, then the solution is to use and implement extension methods of that type.
8. If you create extension methods that have the same signature methods as the type you are extending, then the extension methods will never be called.

How to use Extension Methods :

Example 1:

```
public static class MyExtensions
{
public static int WordCount(this String str){
return str.Split(new char[] { ' ', '.', ',' }).Length; }}
}
class Program{
public static void Main(){
string s = "Dot Net Tricks Extension Method Example";
int i = s.WordCount();
Console.WriteLine(i);}}
```

Example 2:

```
using System;

public static class ExtensionMethods
{
    public static string UppercaseFirstLetter(this string value)
    {
        //
        // Uppercase the first letter in the string.
        //
        if (value.Length > 0)
        {
            char[] array = value.ToCharArray();
            array[0] = char.ToUpper(array[0]);
            return new string(array);
        }
        return value;
    }
}

class Program
{
    static void Main()
    {
        //
        // Use the string extension method on this value.
        //
        string value = "dot net perls";
        value = value.UppercaseFirstLetter();
        Console.WriteLine(value);
    }
}
```

Difference in static method and extension method :

The only difference in the declaration between a regular static method and an extension method is the "this" keyword in the parameter list. If you want the method to receive other parameters, you can include those at the end.

Point to be noted:

An extension method must be defined in a top-level static class.

- An extension method with the same name and signature as an instance method will not be called.
- Extension methods cannot be used to override existing methods.
- The concept of extension methods cannot be applied to fields, properties or events.
- Overuse of extension methods is not a good style of programming.

Benefits of Extension Methods:

- Extension methods allow existing classes to be extended without relying on inheritance or having to change the class's source code.
- If the class is sealed than there in no concept of extending its functionality. For this a new concept is introduced, in other words extension methods.

•	This feature is important for all developers, especially if you would like to use the dynamism of the C# enhancements in your class's design.
•	You can call an extension method in the same way you call an instance method. In Visual Studio, an extension method in IntelliSense has a downward arrow on it. This is a visual clue to how methods are represented.
•	 Extension methods can have many arguments. You can even use variable "params" arguments with extension methods, as with any method. Because extension methods are static methods, there is no significant performance difference.
•	In your code you invoke the extension method with instance method syntax. However, the intermediate language (IL) generated by the compiler translates your code into a call on the static method. Therefore, the principle of encapsulation is not really being violated. In fact, extension methods cannot access private variables in the type they are extending.
•	 You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if a type has a method named Process(int i), and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds.

B. Delegates and Anonymous Methods:

The concept of anonymous method was introduced in C# 2.0. An anonymous method is inline unnamed method in the code. In versions of C# before 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduced anonymous methods and in C# 3.0 and later, lambda expressions supersede anonymous methods as the preferred way to write inline code. However, the information about anonymous methods in this topic also applies to lambda expressions. There is one case in which an anonymous method provides functionality not found in lambda expressions. Anonymous methods enable you to omit the parameter list. This means that an anonymous method can be converted to delegates with a variety of signatures. This is not possible with lambda expressions.

It is created using the delegate keyword and doesn’t required name and return type. Hence we can say, an anonymous method has only body without name, optional parameters and return type. An anonymous method behaves like a regular method and allows us to write inline code in place of explicitly named methods. An anonymous method is a method without a name - which is why it is called anonymous. You don't declare anonymous methods like regular methods.

Where to use Anonymous Methods :

Anonymous methods can be used in the place where there is a use of a delegate. To understand anonymous methods we should understand the usage of delegate first.

Delegate is similar to function pointer in C and C++. A function pointer in C is a variable that points to function. Similarly, a delegate is a reference type in .net that is used to call the methods of similar signature as of the delegate. In C# 1.0, you created an instance of a delegate by explicitly initializing it with a method that was defined elsewhere in the code. C# 2.0 introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 introduced lambda expressions, which are similar in concept to anonymous methods but more expressive and concise. These two features are known collectively as anonymous functions. In general, applications that target version 3.5 and later of the .NET Framework should use lambda expressions. The following example demonstrates the evolution of delegate creation from C# 1.0 to C# 3.0:

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

Why should we use Anonymous method?
We can reduce the code by preventing delegate instantiation and registering it with methods..
It increases the readability and maintainability of our code by keeping the caller of the method and the method itself as close to one another as possible
Key points about anonymous method

1. A variable, declared outside the anonymous method can be accessed inside the anonymous method.
2. A variable, declared inside the anonymous method can’t be accessed outside the anonymous method.
3. We use anonymous method in event handling.

4. An anonymous method, declared without parenthesis can be assigned to a delegate with any signature.
5. Unsafe code can't be accessed within an anonymous method.
6. An anonymous method can't access the ref or out parameters of an outer scope.
7. The scope of the parameters of an anonymous method is the anonymous-method-block.
8. It is an error to have a jump statement, such as goto, break, or continue, inside the anonymous method block if the target is outside the block. It is also an error to have a jump statement, such as goto, break, or continue, outside the anonymous method block if the target is inside the block.
9. The local variables and parameters whose scope contains an anonymous method declaration are called outer variables of the anonymous method. For example, in the following code segment, n is an outer variable: int n = 0;
   Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };
10. A reference to the outer variable n is said to be captured when the delegate is created. Unlike local variables, the lifetime of a captured variable extends until the delegates that reference the anonymous methods are eligible for garbage collection.
11. An anonymous method cannot access the ref or out parameters of an outer scope.
12. No unsafe code can be accessed within the anonymous-method-block.
13. Anonymous methods are not allowed on the left side of the is operator.

C. Lambda Expressions:

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.
To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side. For example, the lambda expression x => x * x specifies a parameter that's named x and returns the value of x squared. You can assign this expression to a delegate type, as the following example shows:

```
delegate int del(int i);
static void Main(string[] args)
{
   del myDelegate = x => x * x;
   int j = myDelegate(5); //j = 25
}
```

Types :

1. **Expression Lambdas.**
2. **Statement Lambdas.**
3. **Lambdas with the Standard Query Operators .**
4. **Type Inference in Lambdas.**

Expression Lambdas :

A lambda expression with an expression on the right side of the => operator is called an expression lambda. Expression lambdas are used extensively in the construction of Expression Trees (C# and Visual Basic). 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 are evaluated outside of the .NET Framework, such as in 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.

Various Examples :

1. s => s.Age > 12 && s.Age < 20
2. (s, youngAge) => s.Age >= youngage; // with multiple parameters.
3. (Student s,int youngAge) => s.Age >= youngage; // with multiple parameters with data types to remove the ambiguity.
4.() => Console.WriteLine("Parameter less lambda expression") // without parameter.

Statement Lambdas:

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");
```

Statement lambdas, like anonymous methods, cannot be used to create expression trees.

Lambdas with the Standard Query Operators :

Many Standard query operators have an input parameter whose type is one of the Func<T, TResult>family of generic delegates. These delegates use type parameters to define the number and types of input parameters, and the return type of the delegate. Func delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the following delegate type:

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

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 parameter's delegate type, and other factors as described in the C# 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<Customer>, 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");
```

The general rules for lambdas are as follows:

- 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.

Note that 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..

Types of LINQ query:

There are two basic ways to write a LINQ query:

1. **Query Syntax or Query Expression Syntax**
2. **Method Syntax or Method extension syntax or Fluent**

1. **Query Expression :** Query expressions **are special statements used for querying a data source using the LINQ. LINQ are just extension methods that you call and returns the data that you want. These methods are located in the System.Linq namespace so you must include it when you want to use LINQ in your project. Query expressions are translated into their equivalent method syntax that can be understood by CLR.**

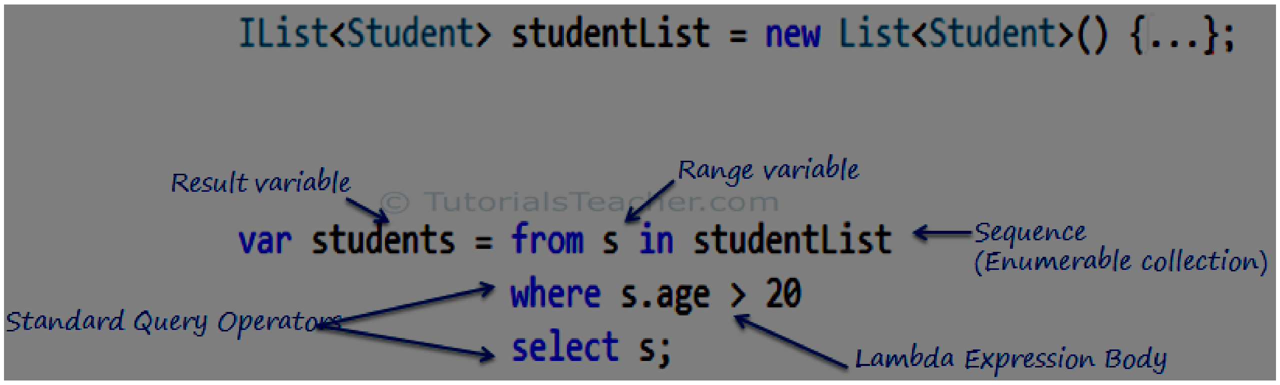
C# is an imperative language which means that you write the step by step codes to make something happen, but LINQ promotes declarative programming. This simply means that you tell the computer exactly what you want and the computer will handle everything else. Before LINQ, you can only use imperative programming for querying results. For example, suppose that you want to get all the even numbers from an array. Without LINQ and using imperative style of programming, your code will look like this:

```
List<int> evenNumbers = new List<int>();
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

foreach(int num in numbers)
{
  if (num % 2 == 0)
    evenNumbers.Add(num);
}
```

Now take a look at the declarative version that uses the query expression syntax :

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
var evenNumbers = from n in numbers
                where n % 2 == 0
                select n;
```



```
IList<Student> studentList = new List<Student>() {...};

var students = from s in studentList
               where s.age > 20
               select s;
```

Query syntax starts with a From clause followed by a Range variable. The From clause is structured like "From rangeVariableName in IEnumerablecollection". In English, this means, from each object in the collection. It is similar to a foreach loop: foreach(Student s in studentList).

After the From clause, you can use different Standard Query Operators to filter, group, join etc. There are around 50 Standard Query Operators available in LINQ. In the above figure, we have used "where" operator (aka clause) followed by a lambda expression body. (We will see lambda expression in detail in the next section.)
LINQ query syntax always ends with a Select or Group clause. The Select clause is used to shape the data. You can select the whole object as it is or only some properties of it. In the above example, we selected the whole student object as it is, but you can also write:select s.StudentName, this will only return Enumerable of StudentName string.
In the following example, we use LINQ query syntax to find out teenager students from the Student collection (sequence).

```
// Student collection
IList<Student> studentList = new List<Student>>() {
     new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
     new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
     new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
     new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
     new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
  };

// LINQ Query Syntax to find out teenager students
var teenAgerStudent = from s in studentList
                  where s.Age > 12 && s.Age < 20
                  select s;
```

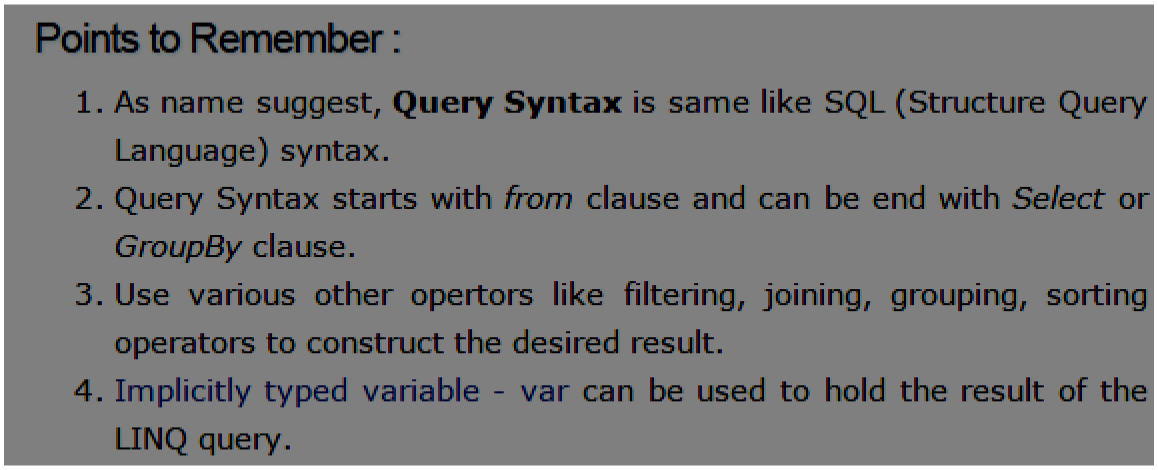
The keywords used in a query expression such as from and select are examples of contextual keywords. They are only treated as keywords during specific events and locations such as a query expression. For example, you can simply use the word select as a variable name if it will not be used in a query expression.

The result of the query is of type IEnumerable<T>. If you will look at our example, the result was placed in a variable of type var which means it uses type inference to automatically detect the type of the queried data. You can, for example, explicitly indicate the type of the query result like this:

```
IEnumerable<int> result = from s in studentList
                 where s.Age > 12 && s.Age < 20
                 select s;
```

but it requires you to know the type of the result in advance. It is recommended to use var instead to take advantage of a lot of its features.

The LINQ has a feature called deferred execution. It means that the query expression or LINQ method will not execute until the program starts to read or access an item from the result of the query. The query expression actually just returns a computation. The actual sequence of data will be retrieved once the user asks for it. For example, the query expression will be executed when you access the results using a foreach loop. You will learn more about deferred execution a little bit later.



Points to Remember :

1. As name suggest, **Query Syntax** is same like SQL (Structure Query Language) syntax.
2. Query Syntax starts with *from* clause and can be end with *Select* or *GroupBy* clause.
3. Use various other opertors like filtering, joining, grouping, sorting operators to construct the desired result.
4. Implicitly typed variable - var can be used to hold the result of the LINQ query.

2. LINQ methods :

LINQ is made possible by the extension methods that are attached to IEnumerable<T> interface. You can call these methods directly, but you need to have a knowledge of lambda expressions. You can also use query expressions which syntax looks like SQL. Query expressions are the main tool you will use to query data using LINQ although you can call the extension methods and use lambda expressions. Method syntax (also known as fluent syntax) uses extension methods included in the Enumerable or Queryable static class, similar to how you would call the extension method of any class.

LINQ is composed of extensions methods that are attached to the IEnumerable<T> interface. These methods exist in the System.Linq namespace and are members of the Enumerable static class. If you can recall, extension methods are special kinds of methods that are use to extend pre-existing types from the .NET class library or to a class in which you don't have access to the source code. For example, you can add a ToTitleCase() method to the System.String type which you can simply call using ordinary strings to change their case style to title case. Let's examine Select<TSource, Tresult>() method from the System.Linq namespace and look at its declaration, you can see that it is attached to the IEnumerable<T> interface.

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```
The first parameter of an extension method determines which type to extend. It is preceded by the this keyword followed by the type to extend and an instance name. You can also see that the return type of this method is IEnumerable<T>. This will allow you to nest or chain method calls as you will see later.
I have said in an earlier lesson that calling the LINQ methods directly requires you to use lambda expressions. Although using anonymous methods are okay, lambda expressions are much simpler and shorter and makes your code more readable. I, therefore, assume that you have a good knowledge of lambda expression before going on with this lesson.
You will now be presented with another way to query data using LINQ, and that is by using the method syntax which is simply calling the LINQ methods directly.
The .NET Framework contains delegate types that can hold methods with a different number of parameters and different return types. Looking at the definition of the Select() method, the second parameter is a generic delegate with a type of Func<TSource,TResult>. The delegate will be able to accept a method that has one parameter of type Tsource and a return type of TResult. For example,Func<string,int> will be able to accept a method that has one string parameter and returns an int.

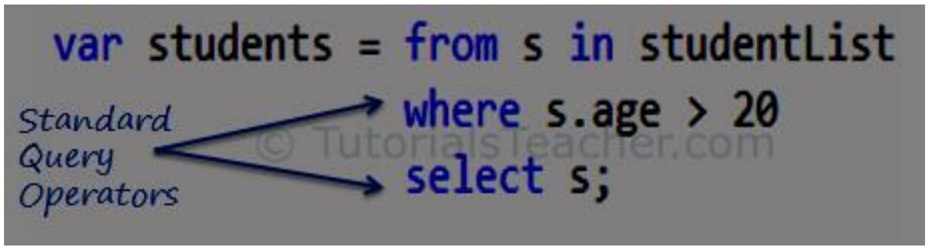
Let's look at how we can use the Select() method by passing a lambda expression as its parameter.
```
int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.Select(n => n);

foreach(var n in result)
{
    Console.Write(n + " ");
}
```
Note that the first parameter of an extension method is not actually a parameter but is used to indicate which type to extend. Therefore, the second parameter which accepts a lambda expression becomes the only parameter of method Select(). Inside the Select()method, we used a lambda expression that accepts one integer parameter and returns an integer value. Again, if you don't know lambda expressions then it might look weird to you. What the lambda expression did was to retrieve every number and then add (return the value) to the query result. The code merely queries every number without modifying them. Let's modify the lambda expression inside theSelect() method to do something more useful.

```
int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.Select(n => n + 1);

foreach(var n in result)
{
    Console.Write(n + " ");
}
```

Chapter – 3 (Standard Query or Linq Operators)

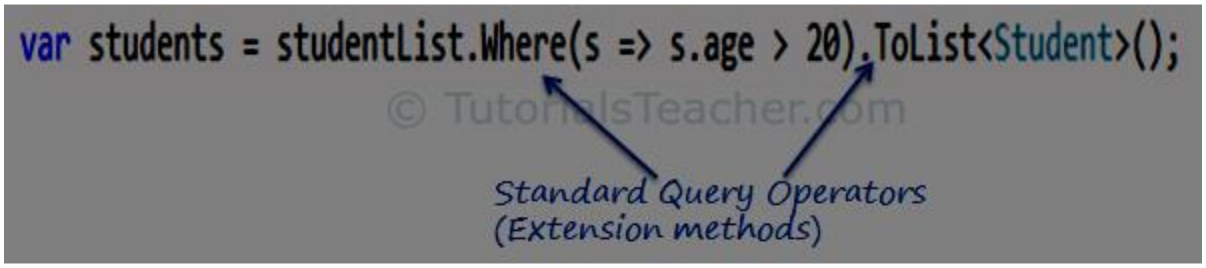
Standard Query Operators in LINQ are actually extension methods for the IEnumerable<T> and Iqueryable<T> types. They are defined in the System.Linq.Enumerable and System.Linq.Queryable classes. There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.

Standard Query Operators in Query Syntax:



```
var students = from s in studentList
               where s.age > 20
               select s;
```

Standard Query Operators in Method Syntax:



```
var students = studentList.Where(s => s.age > 20).ToList<Student>();
```

Standard Query Operators can be classified based on the functionality they provide. The following table lists all the classification of Standard Query Operators:

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

Filtering Operators - Where

The Where operator (Linq extension method) filters the collection based on a given criteria. It accepts a predicate as a parameter.

Where clause in Query Syntax:

```
IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
            new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
      };


var filteredResult = from s in studentList
                              where s.Age > 12 && s.Age < 20
                              select s.StudentName;
```

Where extension method in Method Syntax:

```
var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

Example 1:

The following example uses the Where clause to filter out odd elements in the collection and return only even elements. Please remember that index starts from zero.

```
IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
            new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
      };

var filteredResult = studentList.Where((s, i) => {
                  if(i % 2 ==  0) // if it is even element
                        return true;

            return false;
```

```
        });

foreach (var std in filteredResult)
                Console.WriteLine(std.StudentName);
```

Example 2:

Let's look at an example program that calls the Where extension method from the System.Linq namespace two times with a different lambda expression each time. Lambda expressions use the => operator.

```
static void Main()
        {
                //
                // Example array that contains unwanted null and empty strings.
                //
                string[] array = { "dot", "", "net", null, null, "perls", null };
                //
                // Use Where method to remove null strings.
                //
                var result1 = array.Where(item => item != null);
                foreach (string value in result1)
                {
                        Console.WriteLine(value);
                }
                //
                // Use Where method to remove null and empty strings.
                //
                var result2 = array.Where(item => !string.IsNullOrEmpty(item));
                foreach (string value in result2)
                {
                        Console.WriteLine(value);
                }
        }
```

Example 3:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;
```

```
namespace Operators
{
  class Program
  {
     static void Main(string[] args)
     {
        string[] words = { "humpty", "dumpty", "set", "on", "a", "wall" };

            IEnumerable<string> query = words.Where(w => w.Length > 3);
            foreach (string str in query)
                Console.WriteLine(str);
            Console.ReadLine();
  }
}
```

Multiple Where clause:

You can call the Where() extension method more than one time in a single LINQ query.

```
var filteredResult = from s in studentList
                                  where s.Age > 12
                                  where s.Age < 20
                                  select s;


var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);
```

Points to Remember :

1. **Where** is used for filtering the collection based on given criteria.
2. Where extension method has two overload methods. Use a second overload method to know the index of current element in the collection.
3. Method Syntax requires the whole lambda expression in Where extension method whereas Query syntax requires only expression body.
4. Multiple **Where** extension methods are valid in a single LINQ query.

Filtering Operator - OfType

The OfType operator filters the collection based on the ability to cast to a specified type.

Use OfType operator to filter the above collection based on each element's type

```
IList mixedList = new ArrayList();

mixedList.Add(0);
mixedList.Add("One");
mixedList.Add("Two");
mixedList.Add(3);
mixedList.Add(new Student() { StudentID = 1, StudentName = "Bill" });

var stringResult = mixedList.OfType<string>();
```

Points to Remember :

1. The **Where** operator filters the collection based on a predicate function.
2. The **OfType** operator filters the collection based on a given type
3. **Where** and **OfType** extension methods can be called multiple times in a single LINQ query.

Sorting Operators:

A sorting operator arranges the elements of the collection in ascending or descending order.

Sorting Operator	Description
OrderBy	Sorts the elements in the collection based on specified fields in ascending or decending order.
OrderByDescending	Sorts the collection based on specified fields in descending order. Only valid in method syntax.
ThenBy	Only valid in method syntax. Used for second level sorting in ascending order.
ThenByDescending	Only valid in method syntax. Used for second level sorting in descending order.
Reverse	Only valid in method syntax. Sorts the collection in reverse order.

OrderBy

OrderBy sorts the values of a collection in ascending order by default. Keyword ascending is optional here. Use descending keyword to sort collection in descending order.

```
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var sortedResult = studentList.OrderBy(s => s.StudentName);
```

OrderBy extension method has two overloads. First overload of OrderBy extension method accepts the Func delegate type parameter. So you need to pass the lambda expression for the field based on which you want to sort the collection.

The second overload method of OrderBy accepts object of IComparer along with Func delegate type to use custom comparison for sorting.

The following example sorts the studentList collection in ascending order of StudentName using OrderBy extension method.

OrderByDescending:

OrderByDescending sorts the collection in descending order.

OrderByDescending is valid only with the Method syntax. It is not valid in query syntax because the query syntax uses ascending and descending attributes as shown above.

```
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var result = studentList.OrderByDescending(s => s.StudentName);
```

Sorting on multiple fields:

ThenBy & ThenByDescending

We have seen how to do sorting using multiple fields in query syntax in the previous section.
Multiple sorting in method syntax is supported by using ThenBy and ThenByDescending extension methods.

The OrderBy() method sorts the collection in ascending order based on specified field. Use ThenBy() method after OrderBy to sort the collection on another field in ascending order. Linq will first sort the collection based on primary field which is specified by OrderBy method and then sort the resulted collection in ascending order again based on secondary field specified by ThenBy method.

The same way, use ThenByDescending method to apply secondary sorting in descending order.

The following example shows how to use ThenBy and ThenByDescending method for second level sorting:

```
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },
        new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }
};
var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);
```

Points to Remember :

1. OrderBy and ThenBy sorts collections in ascending order by default.
2. ThenBy or ThenByDescending is used for second level sorting in method syntax.

3. ThenByDescending method sorts the collection in decending order on another field.

4. ThenBy or ThenByDescending is NOT applicable in Query syntax.

5. Apply secondary sorting in query syntax by separating fields using comma.

6. LINQ includes five sorting operators: OrderBy, OrderByDescending, ThenBy, ThenByDescending and Reverse

7. LINQ query syntax does not support OrderByDescending, ThenBy, ThenByDescending and Reverse. It only supports 'Order By' clause with 'ascending' and 'descending' sorting direction.

8. LINQ query syntax supports multiple sorting fields seperated by comma whereas you have to use ThenBy & ThenByDescending methods for secondary sorting.

Grouping Operator: GroupBy & ToLookup

The grouping operators do the same thing as the GroupBy clause of SQL query. The grouping operators create a group of elements based on the given key. This group is contained in a special type of collection that implements an IGrouping<TKey,TSource> interface where TKey is a key value, on which the group has been formed and TSource is the collection of elements that specifies the grouping key value.

Grouping Operators	**Description**
GroupBy	The GroupBy operator returns groups of elements based on some key value. Each group is represented by IGrouping<TKey, TElement> object.
ToLookup	ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate.