5 C# features that will help you to understand LINQ

If this is the first time that you have heard about LINQ, it stands for Language Integrated Query, and it has been introduced in the .NET 3.5 version.

It allows us to query different types of data using the same syntax and it provides us with an intelliSense.

You can write LINQ queries for SQL databases, XML documents, ADO NET datasets and any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface. You can modify the items before you return them. You can also perform calculations on collections, store all or part of the results in a new collection, etc.

The only requirements are:

* to use .NET 3.5 or superior
* to include ***using System.Collections.Generic***at the top of your file.

As a senior software developer, I have been using LINQ for a long time now and I have seen a lot of people struggling with some basic concepts. The truth is that most of us use LINQ without knowing how it is built. The purpose of this post is to understand some of the C# features that make it possible for LINQ to work the way it does.

These features are:

* Lambda Expressions
* Extension methods
* Anonymous types and var keyword
* Query Expression syntax
* Generics

**Lambda expressions**

Let’s say we want to filter a collection of countries and return all the countries in that collection that are in South America. Before lambda expressions were introduced, whenever we needed to pass a function into another method we could use :

* named methods:

The problem with this approach is that it becomes cumbersome to write a named method for every filtering and sorting operation that you have in an application.

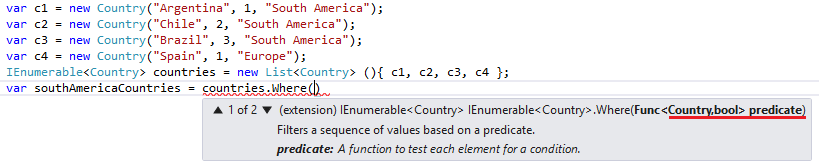
* delegates: to pass anonymous functions. An anonymous function is an “inline” statement or expression that can be used wherever a delegate type is expected.

Using delegates solves the issue of creating named methods each time you want to filter or sort data in the application, but they have a noisy sintax. As you can see in the example, you need to write the word “delegate”, plus the curly braces and the return statement and, what is more, they are difficult to read.

Fortunately, lambda expressions have been introduced in .NET 3.5. They provide a nice and compact syntax to pass an anonymous function into another method.

This example does the same as the previous ones, but it uses lambda expressions.

***Func type:***as we can see in the image below the .where method takes a func as a parameter.



The func type was introduced as an easy way to work with delegates, which allow us to create variables that point to methods.

Func could take from 1 to 17 generic type parameters. When you declare a func type you use generic type parameters to describe the parameters and the return type of a method. The last generic type parameter in a func always describes the return type of a method.

So, for example, we can create a func that takes an integer as a parameter and returns an integer like this.

Or we can create a func that takes two integers as parameters and returns an integer.

when we only have one parameter, we don’t need to add the parenthesis before the lambda operator, but in this example, we have 2 parameters (x and y) so we need to add the parenthesis.

**Extension methods**

Extension methods allow you to extend any existing type, even ones that you haven´t created yourself, by adding additional methods to them.

In other words, extension methods allow us to define methods that appear to be a member of another type, any type, for example, classes, interfaces, structs, even sealed types that we can’t extend using inheritance.

Extension methods are static methods that must be part of a static class and you need to use the “this” modifier in the method declaration to indicate the type you are extending.

In this example, there is a static class called StringExtensions, which holds a static method called Shout. This method takes a string as a parameter. The parameter uses the “this” modifier to indicate the C# compiler that the method is an extension method of the string class.

We can invoke the **.shout()** method from any string, as it was a method from the string class itself.

Behind the scenes, the C# compiler is calling the shout() static method, and passing in the sayHello string as a parameter but the syntax is easier to read than invoking shout() as a static method.

Now that you know a little bit about extensions methods you may have realized how they are related to LINQ.

Basically, LINQ methods are extension methods of the IEnumerable<T> interface, which means that, whenever you have an IEnumerable of any type of object, you can call the LINQ methods.

And because of many of the LINQ extension methods return an IEnumerable<T>, you can chain them together into what is known as a pipeline.

**Anonymous types and var keyword**

The var keyword simply tells the C# compiler to work out what type you’re using by looking at the right-hand side of a variable declaration.

This allows you to write code without repeating yourself.

As you can see in the example above, it is not necessary to write the type of the object in the declaration of it. The C# compiler will automatically infer it.

Also, the **var**keyword allows us to use anonymous types. Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

So for example, I can create an anonymous object which has a Name and a Capital city like this.

Since I am using anonymous types, I didn’t have to create a country class because, under the hood, the C# compiler was able to define a suitable class to make use of. And if a new object with the same properties is created, the C# compiler will be clever enough to use the same anonymous class, so both objects will have the same class.

That means I can put both objects into an array, which the C#compiler will infer to be an array of the anonymous type that was created before.

Something you should know is that if a new object is created and this object contains one extra property, the type of this new object won’t be the same as the one created before. So, it cant be added to the array.

Anonymous objects are useful to LINQ for parsing on information from one stage to the next inside a LINQ pipeline.

**Query expression syntax**

Microsoft has created several new C# keywords (listed below) to allow LINQ statements to be written in a kind of SQL like syntax. This syntax is known as the Query Expression Syntax.

* **from**: Specifies a data source and a range variable (similar to an iteration variable).
* **where:**Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators ( && or || ).
* **select**: Specifies the type and shape that the elements in the returned sequence will have when the query is executed.
* **group**: Groups query results according to a specified key value.
* **into**: Provides an identifier that can serve as a reference to the results of a join, group or select clause.
* **orderby**: Sorts query results in ascending or descending order based on the default comparer for the element type.
* **join**: Joins two data sources based on an equality comparison between two specified matching criteria.
* **let**: Introduces a range variable to store sub-expression results in a query expression.
* **in**: Contextual keyword in a join clause.
* **on**: Contextual keyword in a join clause.
* **equals**: Contextual keyword in a join clause.
* **by**: Contextual keyword in a group clause.
* **ascending**: Contextual keyword in an orderby clause.
* **descending**: Contextual keyword in an orderby clause.

The purpose of this post is not to teach you how to use each of these clauses but to explain how LINQ works. So, I am not going to go into details on how all these keywords are used but I will show you some examples of query expression syntax to discuss it.

As you can observe from the query above, the similarities with SQL are obvious, but something that is important to point out is that this syntax is not only applicable to database data sources. LINQ can work against many different providers, such as Entity Framework or NHibernate.And this makes it really easy to write LINQ queries, which behind the scenes will turn into SQL statements.

As we have seen before, LINQ methods are extension methods of the IEnumerable<T> interface, which means that LINQ can be used anywhere where you have an in-memory collection of objects that implements IEnumerable interface.

**Generics**

Generics gives you the possibility to create methods and classes that are not restricted to a type. I am sure this is a not new concept for you, you have used generics before when you implemented list<T> for example.

This generic class (list<T>) allows you to create a strongly typed list of any sort of objects like int, string or even a list of objects of a class you create.

As it is shown in the example above, you can have a list of whatever type you want, you just need to specify the type between <>.

All the lists, no matter what type of objects they have, will have the same methods. These methods are defined in the definition of the generic class.

To have a better understanding of how generics works let´s check the definition on List, you can do that by right-clicking on the word ***List*** and then pressing ***go to definition.***

As you can see, the definition of the class has a <T>, which means that indicates the type of objects that the collection will have, whenever a method needs to receive an object, for example, the method add, the type of the parameters is represented by T and will be “completed” by the compiler at runtime.

It is thanks to generics that we can use LINQ extension methods no matter what source of data we are working with. All LINQ extension methods are generic functions, so they can be used regardless of which types your collection or lists consists of.

Generics will be useful if you happen to you need to extend LINQ.

LINQ is widely used when you develop a C# application. I cannot think about a single application where I do not have to work with collections whether they are in-memory, XML, files or databases. And the truth is that LINQ is the best way to deal with them.

I hope this post has helped you to clarify some of the C# features that are used by LINQ and, whenever you need to use LINQ, you know exactly why it works the way it does.

Source:

* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq>
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions>
* <https://app.pluralsight.com/library/courses/linq-more-effective/table-of-contents>
* <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/anonymous-types>
* <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords>
* <https://app.pluralsight.com/library/courses/linq-fundamentals-csharp-6/table-of-contents>