Top of Form

LINQ Introduction Part 1 Of 3

Zie ook deze (in Nl): <https://users.hogent.be/de-wolf/csharp/>



[**Sacha Barber**](https://www.codeproject.com/script/Membership/View.aspx?mid=569009)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Rate me!** | |  |  | | --- | --- | | https://www.codeproject.com/script/Ratings/Images/stars-fill-lg.png  https://www.codeproject.com/script/Ratings/Images/stars-empty-lg.png | 4.92 (173 votes) | |  | |

# An introduction to LINQ

Contents

* [Introduction](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#int)
* [What This Article Is Not](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#WhatItsNot)
* [Still Interested ?](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#StillIn)
* [A Little Word About The Attached demo Project](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#DEMO)
* [OK Lets Crack On Then](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#CrackOn)
* [Operator Instructions](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#OpInst)
* [Dynamically Creating Linq Queries](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#DynQry)
* [Thats it](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#ThatsIt)
* [Conclusion](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#Conc)
* [History](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#Hist)
* [Bibilography](https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3#Bibl)

Introduction

LINQ will add new features to both C# and VB.NET. LINQ has three flavours:

* LINQ: Language Integrated Query for in memory objects (LINQ to Objects)
* DINQ: Language Integrated Query for databases (LINQ to ADO NET)
* XLINQ: Language Integrated Query for XML (LINQ to XML)

LINQ is pretty cool, and I have been looking into it as of late, so I thought I would write an article about what I have learned in the LINQ/DLINQ/XLINQ areas, in the hopes that it may just help some of you good folk. This article will be focussed on LINQ, and is the first in a series of 3 proposed articles.

The proposed article series content will be as follows:

* [Part1](https://www.codeproject.com/KB/vista/LINQ_1/aspx) (this article) : will be all about standard LINQ, which is used to query in memory data objects such as List, arrays etc etc
* [Part2](https://www.codeproject.com/KB/vista/LINQ2.aspx) : will be about using DLINQ, which is LINQ for database data
* [Part3](https://www.codeproject.com/KB/vista/LINQ_3.aspx) : will be about using XLINQ, which is LINQ for XML data

But What Is LINQ Anyway?

Well standard LINQ is a new addition to .NET (it adds more dlls basically) that allows the programmer to query inline data as they probably would be used to doing with standard SQL-type syntax.

So where they may have had a query in a database or a SQL query string something like:

Hide   Copy Code

SELECT \* from Books WHERE QuatityInStock > 50 AND Price > 50.00

we would now write the following into as a valid LINQ query (assuming we have the relevant in memory data structure to support the query)

Hide   Copy Code

var result =

from b Books

where b.QuatityInStock > 50 AND Price > 50.00

select b;

See how similar this is. It's very powerful. So thats basically what LINQ allows us to do. And as one can imagine, DLINQ does similar stuff but with database objects, and XLINQ does queries/creation over XML documents.

LINQ also introduces lot of concepts that have really come from other functional programming languages, such as Haskell, LISP. Some of these new concepts are:

* Lambdas (which kind of allow anonymous functions (methods in .NET) to be called over a sequence, a nice source on this is [here](http://www.codeproject.com/csharp/lambdaexpressions.asp))
* Recursive processing over a sequence
* Lazy evaluation

These will hopefully become more familiar to you as we continue.

The main data query sources used for most queries will be simple based on List objects, which contain some really simple class objects. Let's have a look at an example data List objects

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

\_itemList = new List<Item> {

{ ItemID = 1, ItemName = "Enclopedia",

Category="Knowledge", UnitPrice = 55.99M, UnitsInStock = 39 },

{ ItemID = 2, ItemName = "Trainers",

Category="Sports", UnitPrice = 75.00M, UnitsInStock = 17 },

{ ItemID = 3, ItemName = "Box of CDs",

Category="Storage", UnitPrice = 4.99M, UnitsInStock = 13 },

{ ItemID = 4, ItemName = "Tomatoe ketchup",

Category="Food", UnitPrice = 0.56M, UnitsInStock = 53 },

{ ItemID = 5, ItemName = "IPod",

Category="Entertainment", UnitPrice = 220.99M, UnitsInStock

= 0 },

{ ItemID = 6, ItemName = "Rammstein CD",

Category="Entertainment", UnitPrice = 7.99M, UnitsInStock =

120 },

{ ItemID = 7, ItemName = "War of the worlds DVD",

Category="Entertainment", UnitPrice = 6.99M, UnitsInStock =

15 },

{ ItemID = 8, ItemName = "Cranberry Sauce",

Category="Food", UnitPrice = 0.89M, UnitsInStock = 6 },

{ ItemID = 9, ItemName = "Rice steamer",

Category="Food", UnitPrice = 13.00M, UnitsInStock = 29 },

{ ItemID = 10, ItemName = "Bunch of grapes",

Category="Food", UnitPrice = 1.19M, UnitsInStock = 4 }};

\_orderList = new List<Order> {

{ OrderID = 1, OrderName = "John Smith", OrderDate =

DateTime.Now },

{ OrderID = 2, OrderName = "Professor X", OrderDate =

DateTime.Now },

{ OrderID = 3, OrderName = "Naomi Campbell", OrderDate =

DateTime.Now },

{ OrderID = 4, OrderName = "The Hulk", OrderDate =

DateTime.Now },

{ OrderID = 5, OrderName = "Malcolm X", OrderDate =

DateTime.Now }};

So it can be seen that the 1st List simply contains 10 Item objects, and the 2nd List simply contains 10 Order objects. But what do these Item and Order objects look like? As I previously said, they are very simply objects, that are really dumb, and simply there to showcase the talents of what can be done with LINQ.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Collections.Generic;

using System.Text;

namespace LinqApp

{

public class Item

{

#region Instance fields

private int itemID;

private string itemName;

private string category;

private decimal unitPrice;

private int unitsInStock;

#endregion

#region Ctor

/// *<span class="code-SummaryComment"><summary></span>*

/// *ctor*

/// *<span class="code-SummaryComment"></summary></span>*

public Item()

{

}

#endregion

#region Public Properties

public int ItemID

{

get { return itemID;}

set { itemID = value; }

}

public string ItemName

{

get { return itemName;}

set { itemName = value; }

}

public string Category

{

get { return category;}

set { category = value; }

}

public decimal UnitPrice

{

get { return unitPrice;}

set { unitPrice = value; }

}

public int UnitsInStock

{

get { return unitsInStock;}

set { unitsInStock = value; }

}

public string ToString()

{

return "ItemID :" + ItemID + "\r\n" +

"ItemName :" + ItemName + "\r\n" +

"Category :" + Category + "\r\n" +

"UnitPrice :" + UnitPrice + "\r\n" +

"UnitsInStock :" + UnitsInStock;

}

#endregion

}

}

using System;

using System.Collections.Generic;

using System.Text;

namespace LinqApp

{

public class Order

{

#region Instance fields

private int orderID;

private string orderName;

private DateTime orderDate;

#endregion

#region Ctor

/// *<span class="code-SummaryComment"><summary></span>*

/// *ctor*

/// *<span class="code-SummaryComment"></summary></span>*

public Order()

{

}

#endregion

#region Public Properties

public int OrderID

{

get { return orderID;}

set { orderID = value; }

}

public string OrderName

{

get { return orderName;}

set { orderName = value; }

}

public DateTime OrderDate

{

get { return orderDate;}

set { orderDate = value; }

}

public string ToString()

{

return "OrderID :" + OrderID + "\r\n" +

"OrderName :" + OrderName + "\r\n" +

"OrderDate :" + OrderDate;

}

#endregion

}

}

The right hand area of the demo application shows the current query (actual LINQ syntax) and the results obtained.

This is pretty much how all the source data for the demo application is done, there may be some exceptions, where simple arrays of values are used instead of List objects, but I'll mention those when we come to them.

Think of the demo app as a mini LINQ playground.

So that's about all I think you'll need to know about the demo app, for the moment, so shall we continue?

OK Lets Crack On Then

As I previously stated standard LINQ (not DLINQ / XLINQ) operates on in memory data structures such as arrays, collections etc etc. LINQ actually does this using methods known as Standard Query Operators.

The majority of the Standard Query Operators are extension methods that extend IEnumerable<T>.

I think the best way to tackle this subject is to introduce the LINQ Standard Query Operators. And give you a formal definition and an example of each one.

The LINQ specification details the following operators:

1. Restriction operators
2. Projection operators
3. Partitioning operators
4. Join operators
5. Concatenation operator
6. Ordering operators
7. Grouping operators
8. Set operators
9. Conversion operators
10. Equality operator
11. Element operators
12. Generation operators
13. Quantifiers
14. Aggregate operators

Where the Standard Query Operators operate on sequences. Any object that implements the interface IEnumerable<T> for some type T is considered a sequence of that type.

So you can see it's quite some beast. So what I'll attempt to do is give one formal definition and one example for each of the operators. I'll leave further reading (as I am showing one example, there are many possibilities for each operator) as an exercise for the reader.

Operator Instructions

Time for some examples.

## Restriction (WHERE) Operator

The Restriction operator filters a sequence based on a predicate.

Hide   Copy Code

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, int, bool> predicate);

Predicates

What's a predicate you say.

Well Wikipedia says:

"*In formal semantics a predicate is an expression of the semantic type of sets. An equivalent formulation is that they are thought of as indicator functions of sets, i.e. functions from an entity to a truth value.*

*In predicate logic, a predicate can take the role as either a property or a relation between entities."*

And the LINQ Standard Query Operators documentation says:

*"The example below declares a local variable predicate of a delegate type that takes a Customer and returns bool. The local variable is assigned an anonymous method that returns true if the given customer is located in London. The delegate referenced by predicate is subsequently used to find all the customers in London."*

Hide   Copy Code

Func<Customer, bool> predicate = c => c.City == "London";

IEnumerable<Customer> customersInLondon = customers.Where(predicate);

So let's consider this simple example. What we would end up with is an expression that would be something like:

Hide   Copy Code

IEnumerable<Customer> customersInLondon = customers.Where(c => c.City

== "London");

which could also be written in a more conventional way (that those more familiar with SQL would probably like)

Hide   Copy Code

IEnumerable<Customer> customersInLondon =

from c in customers

where c.City == "London"

select c;

we could even be lazier than this, and instead of writing:

Hide   Copy Code

IEnumerable<Customer> customersInLondon =

from c in customers

where c.City == "London"

select c;

we could write:

Hide   Copy Code

var customersInLondon =

from c in customers

where c.City == "London"

select c;

That's how LINQ uses predicates. Basically the easiest way of thinking about what predicates are is to think about them as filters, that will evaluate to True or False, and as such will filter the IEnumerable<T> data source that the Expression is being applied to only contain the elements that match the filter (predicate).

But we digress. We needed to do this once, so that Predicates could be explained. But now that you've all got the hang of that we'll revisit the 1st Standard Query Operator.

Restriction (WHERE) Operator Revisited

Recall the Restriction (Where) Query Operator was defined as:

Hide   Copy Code

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, int, bool> predicate);

The Restriction Query operator can be of either of the forms shown above, where the the first argument of the predicate function represents the element to test. The second argument, if present, represents the zero-based index of the element within the source sequence.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var iSmall =

from it in \_itemList

where it.UnitPrice < 50.00M

select it;

A Little Word About "Var"

One thing that is of interest here, which is the use of the var within this example above. This looks reminiscent of days of old - VB, Flash, JavaScript - basically any not-strongly typed language. And those days were bad. These days we expect and use strongly typed objects. Even better these days we also have Generics, which bring us even more Type control over software we write. Yet here is LINQ code, which is after all new stuff, that will probably part of .NET 3.0. Is this good?

Consider this statement:

*"It is also not required to declare type of query variable, because type inference automatically deduces the type when the var keyword is used."*

Tomas Petricek.

What do we think of this? Well it's certainly better that what VB used to do, which was to determine the type at runtime. What LINQ does it to determine the type at compile time. So used wisely the var type can actually help developers and decrease coding time. Of course if you really want to be a stickler for hardcore typing then what one would have to do something like what is shown below:

**EXAMPLE 1**

Hide   Copy Code

IEnumerable<Item> iSmall =

from it in \_itemList

where it.UnitPrice < 50.00M

select it;

Instead of:

**EXAMPLE 2**

Hide   Copy Code

var iSmall =

from it in \_itemList

where it.UnitPrice < 50.00M

select it;

Can you see the difference?

In the 1st case, we were actually selecting the result type, and it which happens to be of Type Item, so we have to declare the result of the query as IEnumerable<Item> as this matches the query result. This is how to strongly type a query result. If, however, the query was changed to not return an Item Type, say, a string Type then we would need to change the query result type from IEnumerable<Item> to IEnumerable<string> we would have to remember to do this. Also if we had some complicated nested, joined, aggregate (SUM, COUNT) type operators as part of the query, it might be quite a complex type that we have to declare as a return type.

What would you guess the query result type be for this:

Hide   Copy Code

from c in customers

join o in orders on c.CustomerID equals o.CustomerID into co

from o in co.DefaultIfEmpty(emptyOrder)

select new { c.Name, o.OrderDate, o.Total };

or how about this one:

Hide   Copy Code

from c in customers

select new

{

c.CompanyName, YearGroups =

from o in c.Orders

group o by o.OrderDate.Year into yg

select new

{

Year = yg.Key, MonthGroups =

from o in yg

group o by o.OrderDate.Month into mg

select new

{

Month = mg.Key, Orders = mg

}

}

};

See, it gets tricky. We all know typing is good and is our friend. But sometimes it can also be fairly complicated as well.

In the 2nd example above, we simply use var instead of strongly typing the result of the query. This works, and the correct types are inferred, as they would be even for the most complicated query results. Also if we change the query, we don't have to change the var, as it will simply infer the new required types automatically. Job done.

It is personal preference, but var can save time and frustration. Just use it wisely and all should be cool.

## Projection (SELECT) Operator

The Projection operator performs a projection over a sequence.

Hide   Copy Code

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, S> selector);

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, int, S> selector);

The Projection Query operator can be of either of the forms shown above, where the first argument of the selector function represents the element to process. The second argument, if present, represents the zero based index of the element within the source sequence.

So lets see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var iNames = from i in \_itemList select i.ItemName;

And maybe another example:

Hide   Copy Code

var namesAndPrices =

\_itemList.Where(i => i.UnitPrice >= 10).Select(i => new { i.ItemName,

i.UnitPrice }).ToList();

This is an interesting statement. The 1st part is the predicate i => i.UnitPrice >= 10, so only those items with a UnitPrice > 10, are actually selected. Then of those that are selected, we generate a new list (using the ToList() method) which only include ItemName and UnitPrice. Neat huh?

There is also SelectMany, which I have not included here. But after you install LINQ, you can play with this yourself.

## Partitioning (Take / Skip) Operator

The Partitioning operator is made up of four parts:

**Take**

Hide   Copy Code

public static IEnumerable<T> Take<T>(this IEnumerable<T> source,int count);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var MostExpensive2 = \_itemList.OrderByDescending(i => i.UnitPrice).Take(2);

Where this example gets the two most expensive Items.

**Skip**

Hide   Copy Code

public static IEnumerable<T> Skip<T>(

this IEnumerable<T> source,int count);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var AllButMostExpensive2 =

\_itemList.OrderByDescending(i => i.UnitPrice).Skip(2);

Where this example gets all but the two most expensive Items.

**TakeWhile**

The TakeWhile operator yields elements from a sequence while a test is true and then skips the remainder of the sequence. I will leave this as an exercise for the reader.

**SkipWhile**

The SkipWhile operator skips elements from a sequence while a test is true and then yields the remainder of the sequence. I will leave this as an exercise for the reader.

## Join Operator

The Join operator is made up of two parts:

**Join**

Hide   Copy Code

public static IEnumerable<V> Join<T, U, K, V>(

this IEnumerable<T> outer,

IEnumerable<U> inner,

Func<T, K> outerKeySelector,

Func<U, K>> innerKeySelector,

Func<T, U, V> resultSelector);

It looks fairly nasty, but breaking it down a bit, it's really just saying get an outer data source, get an inner data source, get in outer key, get an inner key, and get the resultSet.

So let's see a real world example (using the attached demo project and the \_itemList List and \_orderList local data sources)

Hide   Copy Code

var itemOrders =

from i in \_itemList

join o in \_orderList on i.ItemID equals o.OrderID

select new { i.ItemName, o.OrderName };

Where this example gets all Order objects that have the same OrderID value as that of the current Item.ItemID.

**JoinGroup**

The GroupJoin operator performs a grouped join of two sequences based on matching keys extracted from the elements. I will leave this as an exercise for the reader.

## Concatenation Operator

The Concat operator concatenates two sequences.

Hide   Copy Code

public static IEnumerable<T> Concat<T>(

this IEnumerable<T> first,

IEnumerable<T> second);

So lets see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var items = ( from itEnt in \_itemList

where itEnt.Category.Equals("Entertainment")

select itEnt.ItemName

).Concat

(

from it2 in \_itemList

where it2.Category.Equals("Food")

select it2.ItemName

).Distinct();

Where this example gets all Item objects which have a Category of "Entertainment" and Concatenates that with the result of all Item objects which have a Category of "Food" and ensures there are no duplicates, by using the Distinct() method.

## Order (OrderBy / ThenBy) Operator

The OrderBy / ThenBy family of operators order a sequence according to one or more keys.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

public static OrderedSequence<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector);

public static OrderedSequence<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector, IComparer<K> comparer);

public static OrderedSequence<T> OrderByDescending<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector);

public static OrderedSequence<T> OrderByDescending<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector,

IComparer<K> comparer);

public static OrderedSequence<T> ThenBy<T, K>(

this OrderedSequence<T> source,

Func<T, K> keySelector);

public static OrderedSequence<T> ThenBy<T, K>(

this OrderedSequence<T> source,

Func<T, K> keySelector,

IComparer<K> comparer);

public static OrderedSequence<T> ThenByDescending<T, K>(

this OrderedSequence<T> source,

Func<T, K> keySelector);

public static OrderedSequence<T> ThenByDescending<T, K>(

this OrderedSequence<T> source,

Func<T, K> keySelector,

IComparer<K> comparer);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var orderItems =

\_itemList.OrderBy(i => i.Category).ThenByDescending(i => i.UnitPrice);

Where this example gets all Items objects and simply order them 1st by Category and then by UnitPrice.

## Group (GroupBy) Operator

The GroupBy operator groups the elements of a sequence.

Hide   Copy Code

public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector);

public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector,

IEqualityComparer<K> comparer);

public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(

this IEnumerable<T> source,

Func<T, K> keySelector,

Func<T, E> elementSelector);

public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(

this IEnumerable<T> source,

Func<T, K> keySelector,

Func<T, E> elementSelector,

IEqualityComparer<K> comparer);

public interface IGrouping<K, T> : IEnumerable<T>

{

K Key { get; }

}

So lets see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var itemNamesByCategory =

from i in \_itemList

group i by i.Category into g

select new { Category = g.Key, Items = g };

**NOTE:** This example is quite different from those supplied with the LINQ CTP, and I could not get those to work, as I think the syntax may have changed since Microsoft wrote the LINQ documentation (for example GroupBy did not seem to liked, at least not how they described for this specfic query). The example above does actually work.

This example gets all Item objects and simply groups them by Category. Then it selects the results into a new List (on the fly) where the Category is the key of the group result from the previous step, and the Items is set to be the current value of the items that matched the current grouping in the previous step. A little confusing, but let's have a look at the results that may help a little.

Hide   Copy Code

Category : Knowledge

Enclopedia

Category : Sports

Trainers

Category : Storage

Box of CDs

Category : Food

Tomatoe ketchup

Cranberry Sauce

Rice steamer

Bunch of grapes

Category : Entertainment

IPod

Rammstein CD

War of the worlds DVD

It can be seen that we have all the Item objects from the initial List it's just that now they have been grouped. I grant you it's not as nice as standard SQL syntax, but it does the job.

## Set (Distinct / Union / Intersect / Except) Operators

The Set operators is made up of four parts:

**Distinct**

The Distinct operator eliminates duplicate elements from a sequence.

Hide   Copy Code

public static IEnumerable<T> Distinct<T>(

this IEnumerable<T> source);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var itemCategory = (from i in \_itemList select i.Category).Distinct()

Where this example gets the a unique list of all the Categories for the Items List

**Union**

The Union operator produces the set union of two sequences

Hide   Copy Code

public static IEnumerable<T> Union<T>(

this IEnumerable<T> first,

IEnumerable<T> second);

So let's see a real world example (using the attached demo project and the \_itemList and \_orderList List local data sources)

Hide   Copy Code

var un = (from i in \_itemList select i.ItemName).Distinct()

.Union((from o in \_orderList select o.OrderName).Distinct());

Where this example gets a unique List of all ItemName and then unions this result, with a unique List of all OrderName.

**Intersect**

The Intersect operator produces the set intersection of two sequences.

Hide   Copy Code

public static IEnumerable<T> Intersect<T>(

this IEnumerable<T> first,

IEnumerable<T> second);

So lets see a real world example (using the attached demo project and the \_itemList and \_orderList List local data sources)

Hide   Copy Code

var inter = (from i in \_itemList select i.ItemID).Distinct()

.Intersect((from o in \_orderList select o.OrderID).Distinct());

Where this example gets a unique List of all Item ItemID values and then Intersect this result, with a unique List of all Order OrderID values. The result is a List of ints that are common to both the Item and Order List.

**Except**

The Except operator produces the set difference between two sequences.

Hide   Copy Code

public static IEnumerable<T> Except<T>(

this IEnumerable<T> first,

IEnumerable<T> second);

So let's see a real world example (using the attached demo project and the \_itemList and \_orderList List local data sources)

Hide   Copy Code

var inter = (from i in \_itemList select i.ItemID).Distinct()

.Intersect((from o in \_orderList select o.OrderID).Distinct());

Where this example gets a unique List of all Item ItemID values and then Intersect this result, with a unique List of all Order OrderID values. The result is a List of ints that are common to both the Item and Order List.

## Conversion (ToSequence / ToArray / ToList / ... ) Operators

The Set operators are made up of seven parts:

**ToSequence**

The ToSequence operator returns its argument typed as IEnumerable<T>.

Hide   Copy Code

public static IEnumerable<T> ToSequence<T>(

this IEnumerable<T> source);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var query = \_itemList.ToSequence().Where(i => i.Category.Equals(

"Entertainment"));

Where this example takes a List of Item and converts it to a Sequence and then grabs all elements whos Category is "Entertainment".

**ToArray**

The ToArray operator creates an array from a sequence.

Hide   Copy Code

public static T[] ToArray<T>(

this IEnumerable<T> source);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var query = \_itemList.ToArray().Where(i => i.Category.Equals("Food"));

Where this example takes a List of Item and converts it to a array and then grabs all elements whos Category is "Food".

**ToList**

The ToList operator creates a List<T> from a sequence.

Hide   Copy Code

public static List<T> ToList<T>(

this IEnumerable<T> source);

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var query = \_itemList.ToList().Where(i => i.ItemID > 5).Reverse();

Where this example takes a source List of Item and converts it to a List and then grabs all elements whos ItemID > 5 and reverse it, so higher numbers are at start.

**ToDictionary**

The ToDictionary operator creates a Dictionary<K,E> from a sequence.

Hide   Copy Code

public static Dictionary<K, T> ToDictionary<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector);

public static Dictionary<K, T> ToDictionary<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector,

IEqualityComparer<K> comparer);

public static Dictionary<K, E> ToDictionary<T, K, E>(

this IEnumerable<T> source,

Func<T, K> keySelector,

Func<T, E> elementSelector);

public static Dictionary<K, E> ToDictionary<T, K, E>(

this IEnumerable<T> source,

Func<T, K> keySelector,

Func<T, E> elementSelector,

IEqualityComparer<K> comparer);

So let's see a real world example (using the attached demo project where a simple 2D array is used)

Hide   Copy Code

var scoreRecords = new [] { new {Name = "Alice", Score = 50},

new {Name = "Bob" , Score = 40},

new {Name = "Cathy", Score = 45}

};

var scoreRecordsDict = scoreRecords.ToDictionary(sr => sr.Name);

Where this example creates a new 2D Array and converts that to a key/value Dictionary pair object.

**ToLookup**

The ToLookup operator creates a Lookup<K, T> from a sequence. I'll leave this up to the reader to explore.

**OfType**

The OfType operator filters the elements of a sequence based on a type.

Hide   Copy Code

public static IEnumerable<T> OfType<T>(

this IEnumerable source);

So let's see a real world example (using the attached demo project where a simple array is used)

Hide   Copy Code

object[] numbers = { null, 1.0, "two", 3, 4.0f, 5, "six", 7.0 };

var doubles = numbers.OfType<double>();

Where this example creates a new Array of varying objects, and then the OfType is used to grab only those that are of type Double.

**Cast**

The Cast operator casts the elements of a sequence to a given type. I'll leave this up to the reader to explore.

## Equal Operator

The EqualAll operator checks whether two sequences are equal.

Hide   Copy Code

public static bool EqualAll<T>(

this IEnumerable<T> first,

IEnumerable<T> second);

So let's see a real world example (using the attached demo project and the \_itemList and \_orderList List local data sources)

This is a Not Equal Example:

Hide   Copy Code

var eq = (from i in \_itemList select i.ItemID).Distinct().EqualAll((

from o in \_orderList select o.OrderID).Distinct());

Where this example gets all Item object ItemID and the same for all the Order object OrderID and sees if the entire sequence is equal. They are not, as the Item List contains more elements than the Order List, as such there are certain ItemID that dont appear in the Order List.

This is an Equal Example:

Hide   Copy Code

int[] scoreRecords1 = new [] { 10,20 };

int[] scoreRecords2 = new [] { 10,20 };

var eq2 = scoreRecords1.EqualAll(scoreRecords2);

Where this example creates two new arrays, which have the same elements, and as such when they are compared using the EqualAll they are considered to be equal.

## Element (First / FirstOrDefault / ... ) Operators

The Set operators are made up of nine parts:

**First**

The First operator returns the first element of a sequence.

Hide   Copy Code

public static T First<T>(

this IEnumerable<T> source);

public static T First<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The First operator enumerates the source sequence and returns the first element for which the predicate function returns true. If no predicate function is specified, the First operator simply returns the first element of the sequence.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

string itemName = "War of the worlds DVD";

Item itm = \_itemList.First(i => i.ItemName == itemName);

Where this example takes the 1st element of the List of Item that matches the predicate i.ItemName == itemName where itemName = "War of the worlds DVD".

**FirstOrDefault**

The FirstOrDefault operator returns the first element of a sequence, or a default value if no element is found.

Hide   Copy Code

public static T FirstOrDefault<T>(

this IEnumerable<T> source);

public static T FirstOrDefault<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The FirstOrDefault operator enumerates the source sequence and returns the first element for which the predicate function returns true. If no predicate function is specified, the FirstOrDefault operator simply returns the first element of the sequence.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

string itemName = "A Non existence Element";

Item itm = \_itemList.FirstOrDefault(i => i.ItemName == itemName);

Where this example takes the 1st or default element of the List of Item that matches the predicate i.ItemName == itemName where itemName = "A Non existence Element". Which in this case doesnt match, so we get null returned instead.

**Last**

The Last operator returns the last element of a sequence.

Hide   Copy Code

public static T Last<T>(

this IEnumerable<T> source);

public static T Last<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The Last operator enumerates the source sequence and returns the last element for which the predicate function returned true. If no predicate function is specified, the Last operator simply returns the last element of the sequence.

This works the same way as First, I'll leave this as an excercise for the reader.

**LastOrDefault**

The LastOrDefault operator returns the last element of a sequence, or a default value if no element is found.

Hide   Copy Code

public static T LastOrDefault<T>(

this IEnumerable<T> source);

public static T LastOrDefault<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The LastOrDefault operator enumerates the source sequence and returns the last element for which the predicate function returned true. If no predicate function is specified, the LastOrDefault operator simply returns the last element of the sequence.

This works the same way as FirstOrDefault, I'll leave this as an exercise for the reader.

**Single**

The Single operator returns the single element of a sequence.

Hide   Copy Code

public static T Single<T>(

this IEnumerable<T> source);

public static T Single<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The Single operator enumerates the source sequence and returns the single element for which the predicate function returned true. If no predicate function is specified, the Single operator simply returns the single element of the sequence.

This works the same way as First, I'll leave this as an excercise for the reader.

**SingleOrDefault**

The SingleOrDefault operator returns the single element of a sequence, or a default value if no element is found.

Hide   Copy Code

public static T SingleOrDefault<T>(

this IEnumerable<T> source);

public static T SingleOrDefault<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The SingleOrDefault operator enumerates the source sequence and returns the single element for which the predicate function returned true. If no predicate function is specified, the SingleOrDefault operator simply returns the single element of the sequence.

This works the same way as FirstOrDefault, I'll leave this as an excercise for the reader.

**ElementAt**

The ElementAt operator returns the element at a given index in a sequence.

Hide   Copy Code

public static T First<T>(

this IEnumerable<T> source);

public static T First<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The ElementAt operator first checks whether the source sequence implements IList<T>. If so, the source sequence's implementation of IList<T> is used to obtain the element at the given index. Otherwise, the source sequence is enumerated until index elements have been skipped, and the element found at that position in the sequence is returned.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

Item thirdMostExpensive =

\_itemList.OrderByDescending(i => i.UnitPrice).ElementAt(2);

Where this example orders the sourceList of Item by UnitPrice and then takes the 3rd Item.

**ElementAtOrDefault**

The ElementAtOrDefault operator returns the element at a given index in a sequence, or a default value if the index is out of range.

Hide   Copy Code

public static T ElementAtOrDefault<T>(

this IEnumerable<T> source,

int index);

The ElementAtOrDefault operator first checks whether the source sequence implements IList<T>. If so, the source sequence's implementation of IList<T> is used to obtain the element at the given index. Otherwise, the source sequence is enumerated until index elements have been skipped, and the element found at that position in the sequence is returned.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

Item itm = \_itemList.ElementAtOrDefault(15);

Where this example simple attempts to fetch a non existent element from the List of Item, as such null is returned

**DefaultIfEmpty**

The DefaultIfEmpty operator supplies a default element for an empty sequence.

Hide   Copy Code

public static IEnumerable<T> DefaultIfEmpty<T>(

this IEnumerable<T> source);

public static IEnumerable<T> DefaultIfEmpty<T>(

this IEnumerable<T> source,

T defaultValue);

The DefaultIfEmpty operator allocates and returns an enumerable object that captures the arguments passed to the operator.

This works the same way as FirstOrDefault, I'll leave this as an excercise for the reader.

## Generation (Range / Repeat / Empty ) Operators

The Set operators are made up of three parts:

**Range**

The Range operator generates a sequence of integral numbers.

Hide   Copy Code

public static IEnumerable<int> Range(

int start,

int count);

The Range operator allocates and returns an enumerable object that captures the arguments.

So let's see a real world example (using the attached demo project)

Hide   Copy Code

int[] squares = Sequence.Range(1, 10).Select(x => x \* x).ToArray();

Where this example creates a new array of squared numbers (1 - 10) by using the takes the static Sequence.Range method

**Repeat**

The Repeat operator generates a sequence by repeating a value a given number of times.

Hide   Copy Code

public static IEnumerable<T> Repeat<T>(

T element,

int count);

The Repeat operator allocates and returns an enumerable object that captures the arguments.

So let's see a real world example (using the attached demo project)

Hide   Copy Code

int[] numbers = Sequence.Repeat(5, 5).ToArray();

Where this example creates a new array of 5 repeated 5s Sequence.Repeat method.

**Empty**

The Repeat operator generates a sequence by repeating a value a given number of times.

Hide   Copy Code

public static IEnumerable<T> Empty<T>();

The Empty operator caches a single empty sequence of the given type. When the object returned by Empty is enumerated, it yields nothing.

Im not sure why you would want to do this so I'll leave this as an excercise for the reader.

## Quantifiers (Any / All / Contains ) Operators

The Set operators are made up of three parts:

**Any**

The Any operator checks whether any element of a sequence satisfies a condition.

Hide   Copy Code

public static bool Any<T>(

this IEnumerable<T> source);

public static bool Any<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The Any operator enumerates the source sequence and returns true if any element satisfies the test given by the predicate. If no predicate function is specified the Any operator simply returns true if the source sequence contains any elements.  
The enumeration of the source sequence is terminated as soon as the result is known.

So let's see a real world example (using the attached demo project)

Hide   Copy Code

bool b = \_itemList.Any(i => i.UnitPrice >= 400);

Where this example simply returns a bool if there is any Item with a UnitPrice > 400

**All**

The All operator checks whether all elements of a sequence satisfy a condition.

Hide   Copy Code

public static bool All<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The All operator enumerates the source sequence and returns true if no element fails the test given by the predicate.  
The enumeration of the source sequence is terminated as soon as the result is known.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var itemNamesByCategory =

from i in \_itemList

group i by i.Category into g

where g.All(i => i.UnitsInStock > 0)

select new { Category = g.Key, Items = g };

Where this example uses a group operator, and then uses the All operator to fetch the Items that Match the predicate i.UnitsInStock > 0.

**NOTE:** This example is quite different from those supplied with the LINQ CTP, and I could not get those to work, as I think the syntax may have changed since Microsoft wrote the LINQ documentation (for example GroupBy did not seem to liked, at least not how they described for this specfic query). The example above does actually work.

**Contains**

The Contains operator checks whether a sequence contains a given element.

Hide   Copy Code

public static bool Contains<T>(

this IEnumerable<T> source,

T value);

The Contains operator first checks whether the source sequence implements ICollection<T>. If so, the Contains method in sequence's implementation of ICollection<T> is invoked to obtain the result. Otherwise, the source sequence is enumerated to determine if it contains an element with the given value. If a matching element is found, the enumeration of the source sequence is terminated at that point. The elements and the given value are compared using the default equality comparer, EqualityComparer<K>.Default.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

bool b = \_itemList.Contains(\_itemList[0]);

Where this example simply returns a bool if the source List of Item contains \_itemList[0] which it does.

## Aggregate (Count / LongCount / Sum / ... ) Operators

The Set operators are made up of seven parts:

**Count**

The Count operator counts the number of elements in a sequence.

Hide   Copy Code

public static int Count<T>(

this IEnumerable<T> source);

public static int Count<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The Count operator without a predicate first checks whether the source sequence implements ICollection<t>. If so, the sequence's implementation of ICollection<t /> is used to obtain the element count. Otherwise, the source sequence is enumerated to count the number of elements. The Count operator with a predicate enumerates the source sequence and counts the number of elements for which the predicate function returns true.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var foodCat = (from i in \_itemList

where i.Category.Equals("Food")

select i).Count();

Where this example simply returns a count of all the Item with a Category.Equals("Food").

We must be quite carful with List and Count() as there is also a property within the List class and probably many other collections. So we can't do a query directly on the list when using Count().

What I have done here is a query result, then called the Count() after the original query had returned a result.

**LongCount**

The LongCount operator counts the number of elements in a sequence.

Hide   Copy Code

public static long LongCount<T>(

this IEnumerable<T> source);

public static long LongCount<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

The LongCount operator enumerates the source sequence and counts the number of elements for which the predicate function returns true. If no predicate function is specified the LongCount operator simply counts all elements. The count of elements is returned as a value of type long.

Works the same as Count but it returns a long.

**Sum**

The Sum operator computes the sum of a sequence of numeric values.

Hide   Copy Code

public static Numeric Sum(

this IEnumerable<Numeric> source);

public static Numeric Sum<T>(

this IEnumerable<T> source,

Func<T, Numeric> selector);

The Sum operator enumerates the source sequence, invokes the selector function for each element, and computes the sum of the resulting values. If no selector function is specified, the sum of the elements themselves is computed.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var totals = (from i in \_itemList select i.UnitPrice).Sum();

Where this example simply sums the UnitPrice of all the Item.

**Min**

The Min operator finds the minimum of a sequence of numeric values.

Hide   Copy Code

public static Numeric Min(

this IEnumerable<Numeric> source);

public static T Min<T>(

this IEnumerable<T> source);

public static Numeric Min<T>(

this IEnumerable<T> source,

Func<T, Numeric> selector);

public static S Min<T, S>(

this IEnumerable<T> source,

Func<T, S> selector);

The Min operator enumerates the source sequence, invokes the selector function for each element, and finds the minimum of the resulting values. If no selector function is specified, the minimum of the elements themselves is computed. The values are compared using their implementation of the IComparable<T> interface, or, if the values do not implement that interface, the non-generic IComparable interface.

So let's see a real world example (using the attached demo project and the \_itemList List local data source)

Hide   Copy Code

var minimum = (from i in \_itemList select i.UnitPrice).Min();

Where this example simply fetches minimum UnitPrice of all the Item.

**Max**

The Max operator finds the maximum of a sequence of numeric values.

Hide   Copy Code

public static Numeric Max(

this IEnumerable<Numeric> source);

public static T Max<T>(

this IEnumerable<T> source);

public static Numeric Max<T>(

this IEnumerable<T> source,

Func<T, Numeric> selector);

public static S Max<T, S>(

this IEnumerable<T> source,

Func<T, S> selector);

The Max operator enumerates the source sequence, invokes the selector function for each element, and finds the maximum of the resulting values. If no selector function is specified, the maximum of the elements themselves is computed. The values are compared using their implementation of the IComparable<T> interface, or, if the values do not implement that interface, the non-generic IComparable interface.

Max works the same way as Min, I'll leave it as an excercise for the reader.

**Average**

The Average operator computes the average of a sequence of numeric values.

Hide   Copy Code

public static Result Average(

this IEnumerable<Numeric> source);

public static Result Average<T>(

this IEnumerable<T> source,

Func<T, Numeric> selector);

The Average operator enumerates the source sequence, invokes the selector function for each element, and computes the average of the resulting values. If no selector function is specified, the average of the elements themselves is computed.

Max works the same way as Min, I'll leave it as an excercise for the reader.

**Aggregate**

The Aggregate operator applies a function over a sequence.

Hide   Copy Code

public static T Aggregate<T>(

this IEnumerable<T> source,

Func<T, T, T> func);

public static U Aggregate<T, U>(

this IEnumerable<T> source,

U seed,

Func<U, T, U> func);

The Aggregate operator with a seed value starts by assigning the seed value to an internal accumulator. It then enumerates the source sequence, repeatedly computing the next accumulator value by invoking the specified function with the current accumulator value as the first argument and the current sequence element as the second argument. The final accumulator value is returned as the result.

I have to say this one actually defeated me. I searched and searched for another example, as the LINQ documentation one is pretty dire. Check it out, this one is direct from LINQ documentation.

Hide   Copy Code

var longestNamesByCategory =

products.

GroupBy(p => p.Category).

Select(g => new {

Category = g.Key,

LongestName =

g.Group.

Select(p => p.Name).

Aggregate((s, t) => t.Length > s.Length ? t : s)

});

This is not a very nice example is it. This is kind of what we are getting with LINQ. Its very powerful, but some of it is pure crazy syntax. I mean what the heck is this one above telling someone. It's not very clear to me. Even the fabulous [101 LINQ Samples](http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx) doesnt list an Aggregate operator example. So I guess we'll just have to gloss over this one for the time being.

## Fold

Folding is nice concept straight out of the functional programming world, it allows us to fold in a new function to elements of a list (IEnumerable<T>) in our case. This is very powerful.

Let's have a look at one of these:

Hide   Copy Code

double[] doubles = { 2,4,6,8,10 };

double product = doubles.Fold((runningProduct,

nextFactor) => runningProduct \* nextFactor);

In this simple example, we have an array of which we want to get the product. We can simply use fold to literally fold some inline function (namely runningProduct \* nextFactor) to form the result. Neat huh?

Well thats about it for the Standar Query Operators, if youve made it this far well done. It took me ages to write this, and it's probably taken you ages to read this. So I'll forgive you if you want to come back later. But the next bit is all about dynamically created (at runtime) queries. Up until now it's all been pre-compiled queries, which is all very well but not very realistic. In the real world we would want to do dynamic queries wouldn't we.

# Dynamically Creating LINQ Queries

So far we have looked at static (defined at compile time) queries which is all very well but not really what we wI'll probably want to do for our real world applications.

It is also possible to create LINQ queries programatically using information the user may have entered or selected from a UI. This may be achieved use of the following principles:

By The Use Of Variables

We can simply introduce variables in the query, as shown here:

Hide   Copy Code

decimal priceVal = 50.00M;

*//query will now use a variable so is dynamic*

var iSmall = from it in \_itemList

where it.UnitPrice < priceVal

select it;

So by introducing a simple variable, we can control the query. Quite simple.

QueryExpression Type (Mainly Used In DLINQ)

QueryExpression which according to the [MSDN documentation](http://msdn2.microsoft.com/en-us/library/ms818102.aspx)only has one property called ExpressionOperator which Gets or sets the operator used in the expression.

If, however, we look in Visual Studio 2005 (assuming you have [May 2006 LINQ CTP](http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea&displaylang=en) installed), we get a better picture.

It can be seen from this figure that we can literally provide any ExpressionOperator that we like. This is one part of the secret to creating dynamic LINQ queries at runtime.

So we could actually do something like (paying special attention to the use of the filter, which can be any string)

Hide   Copy Code

*//where filter is a string which could be set to "city = 'London'"*

string filter = "city = 'London'";

expression = QueryExpression.Where(expression,

QueryExpression.Lambda(filter, e));

*//Finally, we create a new query based on that expression.*

*//where db inherits from DataContent and was automatically generated using*

*//the DLINQ tool Sqlmetal.exe*

var query = db.CreateQuery<EmployeeView>(expression);

This example is actually a DLINQ query. But this should be possible in standard LINQ using the Queryable<T> which is new to the March 2007 CTP, though I dont have that CTP installed, and probably wont install it as its 4GB (as its the entire "Orcas" build) and is bound to change again.

The InteractiveQuery project which is installed as part of the May 2006 CTP is a good place to look for dynamic DLINQ queries.

IQuerablable Interface

To do runtime queries over in memory objects (LINQ) you WI'll need the new March 2007 CTP, which allows the user to do this by the use of the NEW Queryable<T> feature.

[The Wayward WebLog](http://blogs.gotdotnet.com/mattwar/archive/2006/05/10/594966.aspx) shows the following example to do this. And I recommend you read this.

Hide   Copy Code

IQueryable q = ...;

ParameterExpression p = Expression.Parameter(typeof(Customer), "c");

LambdaExpression predicate =

QueryExpression.Lambda("c.City = 'London'", p);

Expression where = QueryExpression.Where(q.Expression, predicate);

q = q.CreateQuery(where);

I have since been in contact with Matt Warren author of [The Wayward WebLog](http://blogs.gotdotnet.com/mattwar/archive/2006/05/10/594966.aspx) and he sent me a nice email (see I am trying to help you folks out). So I'll go through what he told me.

If we imagine that we have a method like:

Hide   Copy Code

IQueryable Between(IQueryable query, int min, int max)

{

return from n in query where n >= min && n <= max select n;

}

We now have the facilities to query an IQuerable object. Remember that IQuerable is only available in the new March 2007 CTP though. So let's delve a little deeper. We have this nice method sitting there, that allows us to do some sort of between dynamic query, on an IQuerable object. So let's have a look at how we could call this new method.

Suppose we had a simple array:

Hide   Copy Code

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

and that we then queried this array, like:

Hide   Copy Code

IEnumerable<int> query =

from n in numbers

where n %2 == 1

select n;

OK, so now we have a query result, which is of type IEnumerable<int>. Great. But what we could now do is something like:

Hide   Copy Code

IQueryable<int> query2 = Between(query.AsQueryable(), 0, 10);

So what's going on here? Well we re-use the results of the 1st query, which yielded a IEnumerable<int>, and then we use the .AsQueryable() to get the result from an IEnumerable<int> into an IQuerable object. This is all thanks to the new IQuerable interface, that is now part of the March 2007 CTP. The runtime manages executing IQuerable queries built on top of IEnumerable's.

I'm sure you'll agree we all have to learn how to do this at some stage. Personally I'm going to let the CTP mature a little more and the install instructions become a bit more clear (The current CTP is for the entire "Orcas" project, which is the next version of Visual Studio, so it's huge). But that's only my opinion, if you just cant wait for dynamic queries and a sneaky peak at "Orcas" then download the [March 2007 CTP.](http://www.microsoft.com/downloads/details.aspx?familyid=cf76fcba-07af-47ac-8822-4ad346210670&displaylang=en&tm)

That's It

Well that's actually about it. As I said this article has probably not shown you much that you could not have learned from going to [101 LINQ Samples](http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx) however, all information must come from somewhere. And perhaps some folks would not have known about [101 LINQ Samples](http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx) or even LINQ unless they actually read this article, in which case I've probably done them a favour. Also this article does show actual working code, where as I found that some of the LINQ samples in the LINQ documentation, just did not work, or were so complicated that they would scare some folk. I've really tried to keep the examples in this article as simple as possible.

The next two articles (XLlNQ / DLINQ) should display some new material which should be of use to you all....I Promise I'll try and make them cover new material.

So What Do You Think?

I would just like to ask, if you liked the article please vote for it, as it allows me to know if the article was at the right level or not.

Also, if you think that the next two proposed articles should include this much material or less material. Let me know, after all I want to write articles that actually help people out. I know this one had a lot of stuff in it. But it's new stuff, that is common for LINQ/DLINQ/XLINQ, so lots of information had to be covered. Anyway let me know your thoughts.

Conclusion

I have quite enjoyed constructing this article, and have been quite refreshed at just how easy LINQ is to actually use (well most of it, some of the Group and Aggregate operators are just plain nasty). I also had quite a nostalgic feeling as it reminded me of doing Haskell programming, which I would thoroughly recommend everyone avoid, as its simply crazy. But if you like lambdas, then you should get jiggy with curries and lazy evaluation and all that functional type of stuff. Its quite different actually.

References

1. [Lambda Expressions and Expression Trees: An Introduction](http://www.codeproject.com/csharp/lambdaexpressions.asp)
2. [Lambdas and Curry Notes. From Sussex university (my uni actually)](http://www.informatics.sussex.ac.uk/courses/funcprog/Handouts/lecture5-4.pdf)
3. [Concepts behind the C# 3.0 language](https://www.codeproject.com/KB/vista/csharp3-concepts.asp)
4. [LINQ Project](http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx)
5. [101 LINQ Samples](http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx)
6. [Building Queries](http://msdn2.microsoft.com/en-us/library/aa680438.aspx)
7. [The Wayward WebLog](http://blogs.gotdotnet.com/mattwar/archive/2006/05/10/594966.aspx)
8. [May 2006 LINQ CTP](http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea&displaylang=en)

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)

About the Author



[**Sacha Barber**](https://www.codeproject.com/Members/Sacha-Barber)

<https://www.codeproject.com/Articles/18116/LINQ-Introduction-Part-1-Of-3>