Introduction to LINQ:

Language-Integrated Query

What is LINQ?

- It is a feature introduced in VS 2008

- It is a language extension created for interacting with different data types: it is a declarative extension to an otherwise mostly imperative language.

- **LINQ** provides a way to write compact queries.

- **LINQ** allows to decouple the query structure from the data type

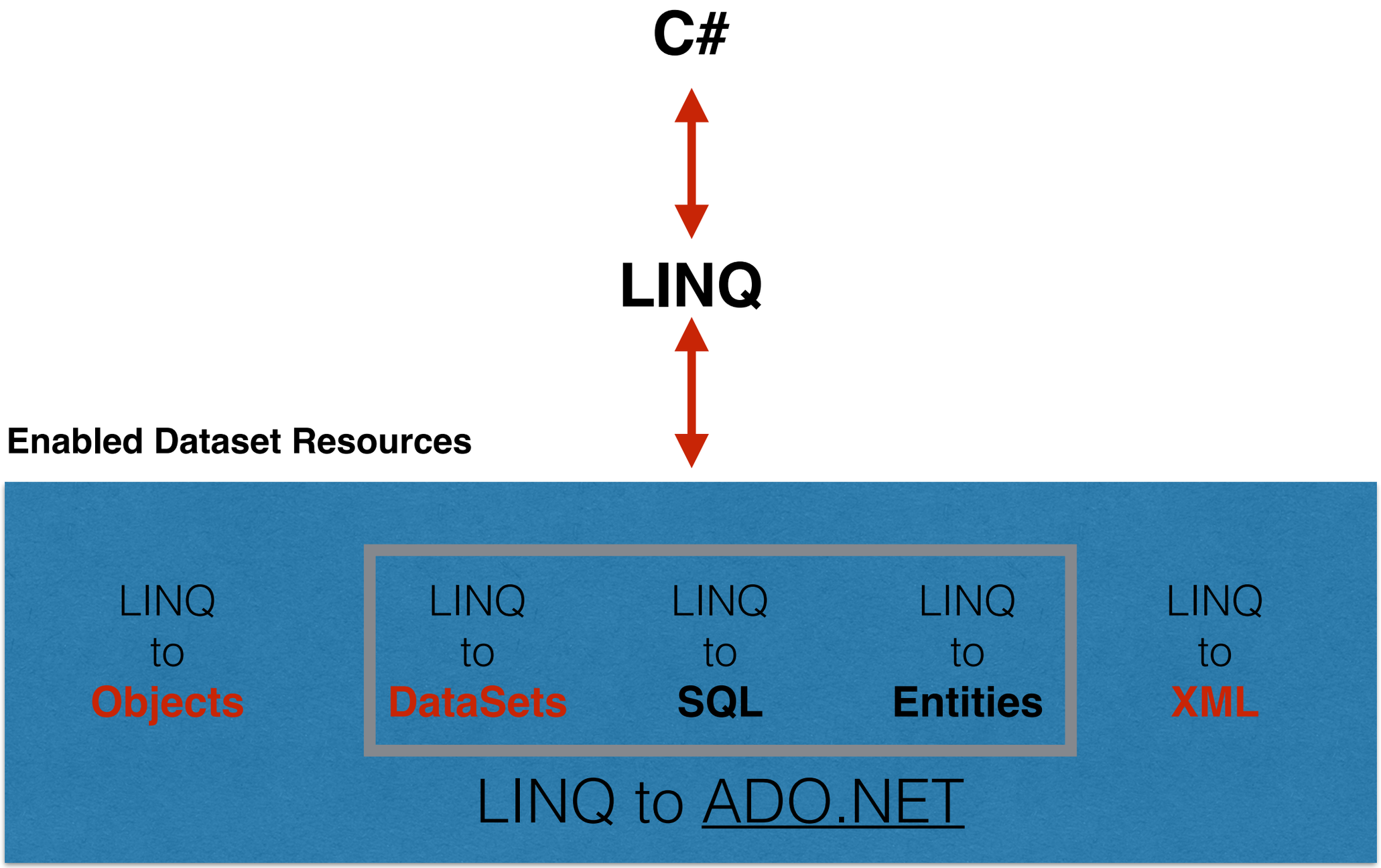
- **LINQ** can also filter and modify the data

- Many data types can now interact with **LINQ**.
- **LINQ** is supported by *IntelliSense*: easier query development

- **LINQ** is supported by the debugger

- **LINQ** is backward-compatible with “old” data types.

LINQ Logical Layout



Other dataset resources are available for connecting to other datasets.
Examples: LINQ to Amazon, LINQ to Excel, LINQ to Flickr, LINQ to Google, …

LINQ Namespaces

System.Linq
Contains the basic classes and interfaces.

System.Linq.Expressions
Contains classes and interfaces for creating LINQ expressions.

System.Data.Linq
Contains classes and interfaces for SQL database interactions.

System.Data.Linq.Mapping
Contains the elements allowing the mapping from/to and imperative language like C# and VB to a declarative language like SQL.

System.Data.SqlClient
Allows connections to SQL Server (MySQLserver needs something else).

System.Xml.Linq
Allows the interaction with data in XML format.

LINQ (Very!) Basic Example of Query Expression

```
//Create the data structure
String[] my_string =
{“Red”,”Black”,”Blue”,”White”};

//Create the LINQ query
var query =
            from a_string in my_string
            select a_string + “\n”;

//Display the result:
foreach (var element in query)
     Console.WriteLine(element);
```

The FROM-IN-SELECT is the simplest query you can build with LINQ

LINQ: Another Basic Example

```
//Create the data structure
String[] my_string =
{“Red”,”Black”,”Blue”,”White”};

//Create the LINQ query
var query =
            from a_string in my_string
            where a_string.Length < 4
            select a_string + “\n”;

//Display the result:
foreach (var element in query)
     Console.WriteLine(element);
```

The role of the IEnumerable Interface

This interface provides access to sequences of items in a collection. The simplest kind of collection is the array.

LINQ uses the functionalities of **IEnumerable** to create queries on collections.

In the previous examples, we used “var” for declaring the query and leaved to the compiler the task of deciding the right type.

Actually, the right type was:

```
IEnumerable<String> query = …. ;
```

Bottom-line: If you create a collection of special objects, be sure to inherit the IEnumerable interface and to implement the required methods for being able to use LINQ on it!

ORDERBY

The data from a query is extracted in the order present in the original data structure. For changing the output order, on can use the **orderby** keyword:

```
var query =
            from a_string in my_string
            where a_string.Length < 4
            orderby a_string
            select a_string + “\n”;
```

Alphabetical sorting

Consecutive orderby:

```
var query =
            from a_string in my_string
            where a_string.Length < 4
            orderby a_string
            orderby a_string.Length
            select a_string + “\n”;
```

First alphabetical sorting

and then sort by length

JOIN

Analogous to the SQL command, **join** allows to combine data from two sources.
The default join is equivalent to the “inner join” of SQL.

```
Int32[] v1 = {3,6,5,4,6,7};
Int32[] v2 = {3,4,6,7,8,9};

var StandardQuery =
                    from Query1 in v1
                    from Query2 in v2
                    where Query1 == Query2
                    select new {Query1 , Query2};
```

Note the definition of the “new” type which is a couple of values.
The result of the join are the couple of numbers present in both arrays at the same time. Equivalently, with a **join**:

```
var JoinQuery =
                    from Query1 in v1
                    join Query2 in v2
                    on   Query1 equals Query2
                    select new {Query1 , Query2};
```

NOTE: You will notice that IntelliSense will suggest you “equals” instead of “==”.

Accessing Multiple Output Fields

```
var JoinQuery =
                from Query1 in v1
                join Query2 in v2
                on   Query1 equals Query2
                select new {Query1 , Query2};


foreach (var output_pair in JoinQuery){
    Console.Write(output_pair.Query1.ToString() +
                  output_pair.Query2.ToString());
}
```

LET

There are cases where you should perform a calculation on a number of fields in a query. Doing such calculation over and over again is inefficient.
The **let** keyword is designed for storing calculation values.

```
Int32[] v1 = {3,6,5,4,6,7};
Int32[] v2 = {3,4,6,7,8,9};

var LetQuery =
                  from  Query1 in v1
                  from  Query2 in v2
                  let   Square = Query1 * Query2
                  where Square > 10
                  select new {Query1 , Query2 , Square};
```

The use of **let** also generally improves the readability of the code.

LINQ and Language Extensions

LINQ supports declarative statements embedded into an imperative language.
For allowing such a possibility, MS provides **language extensions** to C#.

1) The **`var`** keyword:
Allows you to define a variable without specifying the type. The compiler will determine it for you. Note: it is not possible to initialize var variables to null!
"**`var`**" is used to declare variables (see e.g. before with LINQ queries) and within **`foreach`** loops.

2) **Extension Methods:**
Allows you to add new methods to an existing type without deriving a new type.

3) **Object Initializers**:
Allow the simultaneous instantiation and initialization of an object.

4) **Collection Initializers**:
The same as 3) but applied to collections.

5) **Lambda Expressions**
Used in C# for passing an algorithm to a method (seen before).

6) **Query expressions** (seen before) 7) …. more …..

Extension Methods: Basic Example

```
namespace ExtensionMethods
{
    public static class MyExtensions {
        public static int WordCount(this String str){
            return str.Split(new char[] { ' ', '.', '?' },
             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

```
using ExtensionMethods;

string s = "Hello Extension Methods";
int i = s.WordCount();
```

Object Initializers

```
public struct Person
{
    public String Name;
    public String Address;
    public String City;
}

//Instantiate and Initialize:
Person aPerson = new Person
{
    Name= “John Doe”;
    Address = “123-4567 1st Ave”;
    City = “Springfield”;
};
```

Collection Initializers

```
List<String> Names = new List<String>
{
    “Al”,
    “John”,
    “Jack”,
};

List<Person> persons = new List<Person>
{
    new Person {
        Name = “John”,
        City = “Vancouver”l
    } ,

    new Person {
        Name = “Tony”,
        City = “New Westminster”
    }

    ……
}
```

List of standard types

List of new types

Lambda Expressions ("Lambda" operator: =>)

x => y "x goes in y"

```
//Other examples:
x => x.Length

(x,y,z) => x+y / z

(x,y) => {
          if (x>y) return (x);
          else return (y);
        }
```

```
//Use of Lambda expressions:
delegate int del(int i, int j);
static void Main(string[] args)
{
    del aDelegate = x => x * y;
    int output = aDelegate(5,7);
}
```

```
//Use with LINQ
int[] Nums = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
double averageValue = Nums.Where(num => num % 2 == 1).Average();
Console.WriteLine(averageValue);
```

IEnumerable and IEnumerable<T>

We saw already what an interface is and the role of the IEnumerable one.
LINQ uses IEnumerable, or IEnumerable<T> to create an **iterator** for performing queries.

```
String[] mystrings = {“alpha”,”beta”,”gamma”,”delta”,”eta”};

IEnumerable<String> results =
      mystrings.Where(value => value.Length > 4);

foreach (String s in results)
      Console.WriteLine(s + “\n”);
```

If you would like to *avoid a deferred evaluation* of **where**, you can convert the result into and array right away:

```
String[] results =
      mystrings.Where(value => value.Length > 4).ToArray();
```

Query Expressions (seen at the beginning)

Query Expressions are another LINQ language extension which allows to embed declarative statements into C#.
Remember:

- Always begin with the **`from`** keyword
- Contains zero or more **`where`**, **`let`**, **`from`** keywords
- Includes zero or more **`orderby`** (**`ascending`**, **`descending`**) keywords
- Ends with **`select`** or **`group`**
- Can optionally continue with zero or more **`join`** keywords.

The query can be written in SQL-like style (see before) or using the “dot-style” notation.

EXAMPLE:

```
var query =
   my_string.Where(strval => strval.Length>5).Select(
                  my_str => my_str + “\n”);
```

The style of the query depends from the problem at hand and/or from the programmer’s own style

Deferred Operators

Deferred operators are used to interpret the result of a query dynamically.
A list of the available operators is the following:

```
AsEnumerable, AsQueryable,
Cast, Concat,
DefaultIfEmpty, Distinct,
Empty, Except,
GroupBy, GroupJoin,
Intersect, Join,
OfType, OrderBy,
OrderByDescending,
Range, Repeat,
Reverse, Select,
SelectMany, Skip,
SkipWhile, Take.
```

Concat

Allows the concatenation of data from different sources.
EXAMPLE:

```
String[] s1 = {“alpha”,”beta”};
String[] s2 = {“gamma”,”delta”};

var query = s1.Concat<String>(s2);

foreach (String str in query) Console.WriteLine(str+”\n”);
```

Note: **`Concat`** does not remove duplicates.

Filtering with OfType + Where

In some cases, the object you query might contain data of different types.
For selecting only data of a specific type, you can use **OfType**:

```
Dictionary<String, Object> myDict =
                  new Dictionary<String,Object>()

myDict.Add("First",1);
myDict.Add("Second","John");
myDict.Add("Third","Jack");
myDict.Add("Fourth",55);

var query = myDict.Values.OfType<String>();

foreach (String s in query)
   Console.WriteLine(s);
```

Overcoming Empty Data: DefaultIfEmpty

```
List<String> data = new List<String>;

var query = from x in data select x;

foreach (var s in query.DefaultIsEmpty())
      Console.WriteLine(s);
```

The “default” empty answer can be created.

EXAMPLE:

```
public static IEnumerable<String>
      DefaultIsEmpty(this IEnumerable<String> source)
{
   if (source.Count<String>()>0) return sourc;
   else List<String> default = new List<String>;
   default.Add(“Empty”);
   return default;

}
```

Grouping

This is a rather generic example of grouping in a LINQ query:

```
String[] mystrings = {“alpha”,"beta","gamma","delta","eta",
                      “theta","xi","sigma","omicron","omega"};

var query =
          from str in my strings
          group str
          by s.SubString(0,1)
          into Groups
           orderby Groups.Key
           select Groups;

//Print the grouping key
foreach (var group in query) Console.Write(group.Key + “\n”);

//Print the search result
foreach (String in query) Console.WriteLine(s);
```

Distinct, Except, Intersect, Union

```
String[] s1 = {“alpha”,”beta”,”gamma”,”delta”};
String[] s2 = {“alpha”,”delta”,”omega”,”phi”};

var query1 = s1.Distinct();

var query2 = s1.Except(s2);

var query3 = s1.Intersect(s2);

var query4 = s1.Union(s2);
```

Non-Deferred Operators

Non-deferred operators calculate and store the result of a query right away, contrary to the deferred operators which evaluate the query every time the data has changed.
Non-deferred operators are useful in the cases where results must be consistent throughout the program flow/operation.
Non-deferred operators are:

```
Aggregate, All,
Any, Average,
Contains, Count,
ElementAt, ElementAtOrDefault,
First, FirstOrDefault,
Last, LastOrDefault,
LongCount, Max, Min,
SequenceEqual, Single, SingleOrDefault,
Sum, ToArray, ToDictionary,
ToList, ToLookup
```