## LINQ

**\*\*\*DefaultIfEmpty, Any, All, OfType, .Element(s), .Descendents**

**Lambda Expression -** input/return data type are not specified, it will **infer** data types from the delegate's signature

**Event Handler Evolution**

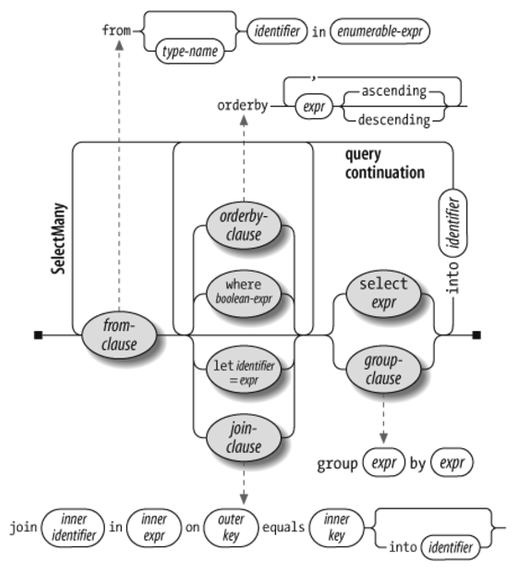
1. button1.Click += new RoutedEventHandler(button1\_Click);
2. button1.Click += button1\_Click;
3. button1.Click += **(o, e) =>  e.something**;

**Expression Tree**:

1+ query operators' lambda expression represented in a tree format to provide a simultaneous evaluation of all the lambda expressions in a single query.

**Query Expression**

**from** a **in** B  
    **where** a.x = 3 **order by** a.y **desc**  
    **select** a;



**Extension Method**

**static** IEnumerable<TSource> **Where**<TSource> ( **this** IEnumerable<TSource> source, **Func**<TSource,bool> predicate)   
**static** IEnumerable<TSource> **OrderBy**<TSource,TKey> ( **this** IEnumerable<TSource> source, **Func**<TSource,TKey> keySelector)   
**static** IEnumerable<**TResult**> **Select**<TSource,TResult> ( **this** IEnumerable<TSource> source, **Func**<TSource,TResult> selector)  // note unlike Where & OrderBy, Select can return a **result of a different Type 'TResult'**

// using the **instance** method  
var t = names.**Where**(n => n.Length >= 2);

**Deferred Query Operators**: those returns IEnumerable<T> or IOrderedEnumerable<T> or IQueryable<T>.

A deferred execution query is **reevaluated** when you **enumerate** it.

**Non-Deferred**: Aggregator, return single element, non-IEnumerable e.g. ToArray, ToList, Count …  
**Outer Variable** -aka Closures

lambda expression reference **local/parameter** variables, these variables are captured. The variable's **value at the time the query is executed**—not at the time the variable is captured.

e.g.

int[] numbers = {1, 2};   
int factor = 10; // **factor is captured below**   
var query = numbers.Select(n => n \* factor);   
factor = 20; // change the capture variable **after composing query but before the execution**.   
foreach(int n in query) Console.Write( n + "|"); // result 20|40 instead of 10|20   
  
for more about closure: <http://blogs.msdn.com/ericwhite/archive/2008/09/12/closures.aspx>

**Subqueries**

A subquery is privately scoped to the enclosing expression and is able to reference the outer lambda argument (or iteration variable in comprehension syntax). i.e. **BOTH names and n**

IEnumerable<string> comprehension = from n in **names**

where n.Length == **(from n2 in names orderby n2.Length select n2.Length).First()**

select n;

**Join**

* Inner Join
  + IEnumerable<V> **Join**<T, U, K, V>(this IEnumerable<T> **outerSequence**, IEnumerable<U> **innerSequence**, Func<T, K> **outerKeySelector**, Func<U, K> **innerKeySelector**, Func<T, U, V> **resultSelector**);
  + Lambda Expression
    - var result = books // **outer sequence**  
          .Join(       
          orders, **// inner sequence**  
          b => b.id, **// outerKeySelector**  
          o => o.bookId, **// innerKeySelector**  
          (b, o) => new { ... }); **// resultSelector**
  + Query Expression
    - var result = from b in dc.books **// outer sequence**  
          join o in dc.orders **// inner sequence**  
          **on** b.id **// outerKeySelector**  
          **equals** o.bookId **// innerKeySelector**  
          select new { ... } **// resultSelector**
* **Outer Joins pp. 435**

**"let** *keyword*" - introduces a new **local variable** alongside the iteration variable. e.g.

let

IEnumerable<string> query =

    from n in names

**let *vowelless* = Regex.Replace (n,"[aeiou]", "")**

    where ***vowelless***.Length > 2

    orderby vowelless

    select **n or vowelless**; // Thanks to **let**, n is still in scope.

The approach is particularly advantageous in this example because it allows the

**Conditionally Building Queries** // note: we use the **explicit method syntax** instead of a query expression.

**IEnumerable<Book> query;**  
query = SampleData.Books;  
*if (minPageCount.HasValue)*                                      
   **query = query**.Where(book => book.PageCount >= minPageCount.Value);  
*if (!String.IsNullOrEmpty(titleFilter))*                           
   **query = query**.Where(book => book.Title.Contains(titleFilter));  
*if (sortSelector != null)*               
   **query = query**.OrderBy(sortSelector);

#### Query Syntax Versus Lambda Syntax

***Query syntax*** is better using:

* ***Let***A clause for introducing a new variable alongside the iteration variable
* *SelectMany*, *Join*, or *GroupJoin* followed by an outer iteration variable reference

***Lambda syntax*** is shorter when queries use only a **single operator**, .

Finally, there are many operators that have no query expression keyword. These require that you use lambda syntax—at least

#### Mixed Syntax Queries e.g.

int count = (from **name in names**  
 **where n.Contains ("a")**  
 **select name**  
 )**.Count( )**;

**SingleOrDefault vs FirstOrDefault**

**DefaultIfEmpty**(a default instance)

If empty, returns a single item Collection which the item is the specified default instance.

**Distinct**: return distinct elements from a squence. returns IEnumerable<T>

**Any**: determine whether ANY elements in the sequence satisfy a condition

**bool** exists = orders.**Any**(o => o.price > 50.0); // Any order in the orders has price > $50

OR bool exists = orders.Any();

**All**: determine whether ALL elements in the sequence satisfy a condition

**bool** exists = orders.**All**(o => o.price > 50.0); // All order in the orders has price > $50

**TakeWhile -** operator enables you to take elements from a collection as long as a given condition is true

**Average/Count/Min/Max/Sum**

**OfType<TResult>**: finds elements of only the given type TResult from a collection that has elements of several types

**object[]** things = {"Sam",1,DateTime.Today,"Eric"};

things.**OfType<string>()**.ToList();

**SelectMany – flattens a list of lists – i.e. replace nested loops using SelectMany**

public class **PhoneNumber** {

public string Number { get; set; }

}

public class Person {

public **IEnumerable<PhoneNumber>** PhoneNumbers { get; set; }

public string Name { get; set; }

}

IEnumerable<Person> people = new List<Person>();

// Select gets **a list of lists** of phone numbers

**IEnumerable<IEnumerable<**PhoneNumber>> phoneLists = people.Select(p => p.PhoneNumbers);

// SelectMany **flattens** it to **just a list of** phone numbers.

**IEnumerable<**PhoneNumber> phoneNumbers = people**.SelectMany**(p => p.PhoneNumbers);

// **And to include data from the parent in the result**, pass an expression to the **second parameter (resultSelector)** in the overload:

var directory = **people.**SelectMany(p => p**.PhoneNumbers**,

(**parent, child**) => **new { parent.Name, child.Number }**);

**Group By**

public class Person {

public int PersonId;

public string car;

}

List<Person> persons = new List<Person>();

persons.Add(new Person { PersonId = 1, car = "Ferrari" });

persons.Add(new Person { PersonId = 2, car = "Audi" });

persons.Add(new Person { PersonId = 1, car = "BMW" });

**IEnumerable<IGrouping<int, Person>**> results =

from p in persons

***group*** p ***by*** p.PersonId **into igroupingItem // p.s. igroupingItem is like { p.PersonId (*by*), p (*group*)}**

**select igroupingItem;**

foreach (var **groupingItem** in results) {

foreach (var item in **groupingItem**) {

Console.WriteLine($"{item.PersonId} {item.car}");

}

}

**MoreLinq -** [**https://github.com/morelinq/MoreLINQ**](https://github.com/morelinq/MoreLINQ)

**LINQ To XML** - Emphasis to element not document!

**Constructor**: XObject is one of the XObject derived classes: XElement, XText, XComment, XAttribute, XNode etc...

    XObject o = **new XObject**(objectName, **new XObject**(objectName, new XObject(objectName, new XObject(...

Adding **Namespace**-2 ways

XElement ctor is expecting XName. XName overload the + operator to populate its 2 properties: Namespace & LocalName

* XElement xBook = new XElement(**"{http://www.tubegotchi.com}"** + "Book");
* XNamespace **ns**= "http://www.tubegotchi.com";  
  XElement xBook = new XElement(**ns+** "Book");

Adding Namespace **Prefix** - use XAttribute

**Example: adding default & extended namespaces, plus using prefix:**  
            **XNamespace** default\_ns = "http://www.tgc.com"; // **setup namespace**  
            **XNamespace *extended\_ns*** = "http://www.eurekr.com"; // **setup more namespace**  
  
            XElement xelem = new XElement(default\_ns+ "TubeGotChi",  
                new XAttribute(**XNamespace.Xmlns** + "**myPrefix**", ***extended\_ns***), // **namespace TO prefix**  
                new XElement(new XElement(default\_ns + "X1", "very")),  
                new XElement(new XElement(***extended\_ns*** + "X2", "funny")) ); // **Using prefix**  
  
        Result:

<TubeGotChi **xmlns:myPrefix**="http://www.eurekr.com" xmlns="http://www.tgc.com">  
  <X1>very</X1>  
  <**myPrefix:**X2>funny</extns:X2>  
</TubeGotChi>

**Node Value**

* xBook.ToString(): children include with XML
* (string)xBook: children included without XML

**I/O**

* **instance** method: xBook.**Save**(path);
* **static** method: XElement.**Load**(new Uri("..."));
* **static** method: XElement.**Parse**(string xml);

**Query - all return IEnumerable<T>**

* .**Elements**([XName filter]): returns just **immediate Children**, no grand kids. optional params to match a given name
* **.Element**([XName filter]): returns the 1st matching element. same as .Elements.FirstOrDefault
* .**Ancestors**([XName filter]), .**AncestorsAndSelf**
* .**Descendants**([XName filter]), .**DescendantsAndSelf** - return "Descendants" including Children & Grand Kids and so on.
* .**DescendantNodes**([XName filter]), .**DescendantNodesAndSelf** - return IEnumerable<XNode>, i.e. ANY X types. e.g. XComment
* .**Attributes**: return IEnumerable<XAttribute>  **n/v pair**

e.g. xBook.Descendants("an Element Name").Where(e => ...).OrderBy(...)...

**Class Methods vs Extension Methods (Operator)**

* p.s. XContainer derived from XNode!
* **Class** Methods - returns Elements or Descendents of **THIS XContainer**
  + query against an **XContainer** derived type. e.g. XElement/XDocument/XNode.
  + IEnumerable<XElement> **XContainer**.Elements()
  + e.g. ***xElement*.Elements**("xyz");
  + p.s. XContainer doesn't derived from IEnumerable!
* **Extension** Methods - return Elements or Descendants of a **IEnumerable<T> Sequence**
  + e.g. IEnumerable<XElement> Elements<T>(this **IEnumerable<T>** source) where **T : XContainer**
  + e.g. xDocument.***Descendants*().Elements**("xyz");

***XElement* Add/Delete/Update**

**Add**:

1. ctor
2. xBook.**AddFirst**/**Add** - current element's **Children**
3. xBook.**AddBeforeSelf**/**AddAfterSelf** - current element **Sibling**

**Delete**

xBook.**Remove** - remove the **current** node from parent.

**Update**

xBook.**Value** - **get**/**set** the text content.

**Unification** of ***Child* Add/Update/Delete**

xBook.**SetElementValue**("a XName type name", "a Object type value")

***XAttribute* Add/Delete/Update**

**Add**

**xBook**.**Add**(new **XAttribute**("name","value"));

**Update**

xAttribute.**Value**

**Delete**

xAttribute.**Remove** - remove the **current** attribute from parent

**Unification** of **Add/Update/Delete**

xBook.**SetAttributeValue**("a XName type name", "a Object type value");

**SetElement/AttributeValue Usage**

1. If **found** the XName param, it will **Update**
2. If **NOT** found, it will **Add**
3. If **found** but **value** is set to **null**, it will **Remove**
4. p.s. It will only affect the **1st child XElement** if more than 1 is found.

OR without Cast by explicitly cast the iteration variable to **Book**.

var query =  
  from **Book** book in books  
  where book.PageCount > 150  
  select new { book.Title, book.Publisher.Name };

**Custom Sort**

void CustomSort<TKey>(Func<Book, TKey> selector, Boolean ascending)  
{  
  IEnumerable<Book> books = SampleData.Books;  
  books = ascending ? books.OrderBy(selector)  
                    : books.OrderByDescending(selector);  
  ObjectDumper.Write(books);

}

For your reference, here is the signature of the OrderBy operator:  
OrderedSequence<TElement> OrderBy<TElement, TKey>(  
  this IEnumerable<TElement> source, Func<TElement, TKey> keySelector)

The additional ascending parameter allows us to choose between the OrderBy  
and OrderByDescending operators. It then becomes possible to use the following  
call to sort using a descending order instead of the default ascending order:  
CustomSort(book => book.Title, false); // selector is the lambda expression book => book.Title