|  |
| --- |
| Reference |
| <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords> |
| <https://docs.microsoft.com/en-us/aspnet/web-forms/videos/data-access/> |
|  |

**LINQ – Language Integration Query**

**Background**:

Earlier, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you must learn different queries languages for each type of data like, SQL Database, XML documents, in memory objects etc.

LINQ provides a consistent query experience for

* Linq to objects
* Linq to sql
* Linq to xml

We can write linq queries in C# for sql server database, XML documents, ADO.NET database and any collection of objects that supports **IEnumerable** or generic **IEnumerable<T>** interface.

* LINQ queries
  + Three parts of query operations
    - Obtain the Data Source
    - Create the Query
    - Execute the Query

|  |
| --- |
| // Defining the Query expression  IEnumerable<Employee> employees = (from emp in dbContext.Employees  where  emp.JobTitle == "Design Engineer"  && emp.Gender =="F"  select emp  );  // Execute the query  foreach (var item in employees)  {  Debug.WriteLine("Data: " + item.NationalIdnumber);  } |

* GroupBy
  + GroupBy clause enable you to group your result based on the key you specify.

|  |
| --- |
| //group by  var employees1 = ( from emp in dbContext.Employees  .AsEnumerable()  group emp by emp.JobTitle into empGroup  where empGroup.Count() >2  orderby empGroup.Key  select empGroup  );  foreach (var item in employees1)  {  Debug.WriteLine("item: " + item.Key);  } |

Sql generated with above query expression

|  |
| --- |
| SELECT [e].[BusinessEntityID], [e].[BirthDate], [e].[CurrentFlag], [e].[Gender], [e].[HireDate], [e].[JobTitle], [e].[LoginID], [e].[MaritalStatus], [e].[ModifiedDate], [e].[NationalIDNumber], [e].[OrganizationLevel], [e].[rowguid], [e].[SalariedFlag], [e].[SickLeaveHours], [e].[VacationHours]  FROM [HumanResources].[Employee] AS [e] |

**Note**

* The resultant sql is not containing any GROUP BY sql clause.
* Because, using group by in LINQ, the result set take in the form of list of lists. Where each element in the list is an object has **key** member.

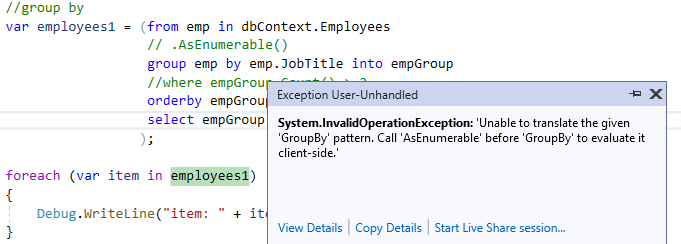
**Case Study# Group By**

* Write the below query without using. AsEnumerable() or .ToList()
* It is going to through runtime exception saying “**unable to translate the given groupBy by pattern call AsEnumerable before groupby to evalute it client-side**”
* Here, it means that **groupBy** operation is going to perform on Client-Side but not on server side.

|  |
| --- |
| //group by  var employees1 = ( from emp in dbContext.Employees  .AsEnumerable()  group emp by emp.JobTitle into empGroup  where empGroup.Count() >2  orderby empGroup.Key  select empGroup  );  foreach (var item in employees1)  {  Debug.WriteLine("item: " + item.Key);  } |

**Deferred Execution**

* The query variable itself will only execute the commands. The actual execution of the query is deferred until you iterate over the query variable like in above for each loop.



**Forcing immediate execution**

* If we want to perform any aggregate functions over the range of the query, we can use aggregate function such as Count, Max, Average, First etc.
* Note, using above aggregate function query will return a single value.
* If we want to return a list of values we can use .ToList() or .ToArray() methods.

**Sorting Data**

|  |  |
| --- | --- |
| OrderBy | Sort values in asc |
| OrderByDescending | Sort values in descending order |
| ThenBy | Performs a secondary sort in ascending order |
| ThenByDescending | Performs a secondary sort in descending order |
| Reverse | Reverser the order of the elements in a collection |

**Set Operations**

Set operations in LINQ refers to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collection.

|  |  |
| --- | --- |
| Distinct or DistinctBy | Removes the duplicate values from a collection |
| Except or ExceptBy | Return the set difference, which means the element of one collection that do not appear in second collection |
| Intersect or IntersectBy | Return the set intersection, which means elements that appears in each of two collections |
| Union or UnionBy | Returns the set union, which means unique elements that appears in either or two collections |

**Quantifier Operations**

* These operators return the Boolean values, indicating some or all the values satisfies the conditions.

|  |  |
| --- | --- |
| All | Determines whether **all** the elements in the sequences satisfy a condition |
| Any | Determines whether **any** of the elements in the sequences satisfy aconditions |
| Contains | Determines whether sequences **contain** the given elements |

**Projection Operators**

* Projects refers to the operations of transferring an object into new form.

|  |  |
| --- | --- |
| Select | Project values that are abased on the transform function |
| SelectMany | Project sequence of values that are based on transform function and then flattens then into one sequence |
| Zip | Produce a sequence of tuples where elements from 2-3 specified sequences  In general, when we want to use **key**,**value** pair programming, we can use this operator. |

**Select**

* Following program demonstrate to project first letter of each string

|  |
| --- |
| // Demonstrate Select clause  List<string> words = new() { "an", "apple", "a", "day" };  var query = from word in words  select word.Substring(0, 1);  foreach (string s in query)  {  Debug.WriteLine(s);  } |

**selectMany**

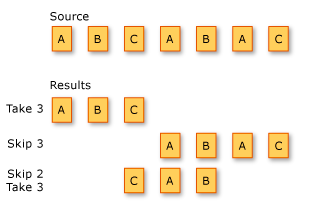
|  |
| --- |
| // Demonstrate Select Many clause (below query uses multiple from clauses  List<string> phrases = new() { "An Apple a Day ", "Quick Brown Fox" };  var tData = from phrase in phrases  from word in phrase.Split(' ')  select word;  foreach (var item in tData)  {  Debug.WriteLine(item);  } |

**Zip**

|  |
| --- |
| IEnumerable<int> numbers = new[] { 1, 2, 3, 4, 5, 6 };  IEnumerable<string> stringValues = new[] { "Apple", "Boy", "Cat", "Dog", "Elephant", "Fish" };  // Expected o/p: (key: value) pairs  foreach ((int key, string value) in numbers.Zip(stringValues))  {  Debug.WriteLine($"Key {key} : Value {value}");  }  **o/p**  Key 1 : Value Apple  Key 2 : Value Boy  Key 3 : Value Cat  Key 4 : Value Dog  Key 5 : Value Elephant  Key 6 : Value Fish |

**Partitioning data**

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.



|  |  |
| --- | --- |
| Skip | Skip elements up to the specified position in the sequence |
| SkipWhile | Skip elements based on predicate function until an element does not satisfy the condition |
| Take | Takes an element up to the specified position in the sequence |
| TakeWhile | Takes elements based on predicate function until an element does not satisfy the condition |
| Chunk | Splits the elements of a sequence into chunks of a specified maximum size |

|  |
| --- |
| Debug.WriteLine("Original Sequence: "); //Expected output: 1,2,3,4,5,6,7,8,9,10  foreach (var item in Enumerable.Range(1, 10))  {  Debug.Write(item + ",");  }  Debug.WriteLine("Skip first two elements "); //Expected output: 3,4,5,6,7,8,9,10  foreach (var item in Enumerable.Range(1, 10).Skip(2))  {  Debug.Write(item + ",");  }  Debug.WriteLine("Take first two elements "); //Expected output: 1,2  foreach (var item in Enumerable.Range(1, 10).Take(2))  {  Debug.Write(item + ",");  }  Debug.WriteLine("Take second Third highest element "); // Expected output: 8  foreach (var item in Enumerable.Range(1, 10).OrderByDescending(a=>a).Skip(2).Take(1))  {  Debug.Write(item);  } |

**Note**: All the above partitioning operations are performed on client side.

**Join Operations**

* A join of two data sources is the association of objects in one data source with objects that share a common attribute in another data source

|  |  |
| --- | --- |
| Join | Joins two sequence based on key selector function and extracts pair of values |
| GroupJoin | Join two sequences based on key selector function and groups the resulting matches for each element |

**Grouping Data**

* Group refers to the operation of putting data into groups so that the element in each group share a common attribute.

|  |  |
| --- | --- |
| GroupBy | Groups elements that shares a common attribute. Each group is represented as  IGrouping<T(Key),Value> object |
| ToLookUp | Inserts elements into a Lookup<Key,Value> (a one to many dictionary) based on key selector function |

**Generation Operators**

Generation refers to creating a new sequence of values

|  |  |
| --- | --- |
| DefaultIfEmpty | Replace an empty collection with a default value singleton collection |
| Empty | Returns an empty collection |
| Range | Generate a collection that contains a sequence of numbers |
| Repeat | Generate a collection that contains one repeated values |

**Equality Operations**

Two sequences who corresponding elements are equal and which have the same number of elements are considered equal

|  |  |
| --- | --- |
| SequenceEqual | Determine whether two sequence are equal by comparing elements in a pair-wise manner |

**Element Operations**

Element operations return a single, specific element from a sequence

|  |  |
| --- | --- |
| ElementAt | Returns an element at a specified index in a collection |
| ElementAtOrDefault | Return an element at a specified index in a collection or a default value if the index is out of range |
| First | Returns the first element of a collection, or the first element that satisfied a condition. |
| FirstOrDefalut | Returns the first element of a collection, or the first element that satisfied a condition. Returns a default value if no such element found |
| Last | Returns the last element of a collection, or the last element that satisfied a condition |
| LastOrDefault | Returns the last element of a collection, or the last element that satisfied a condition, returns a default value if no such element found |
| Single | Returns the only element of a collection or the only element that satisfies a condition |
| SingleOrDefault | Returns the element of a collection or the only element that satisfied a condition. Returns a default value if there is no element to return. |

**Converting Data Types**

* Converting methods changes the type of input objects

**Methods**

* The conversion methods in the below table start with
  + “**As**” methods which start with “As” changes the static type of the source but do no enumerate it
  + “**To**” methods which start with “To” enumerates the source collection and put the items into the corresponding collection type

|  |  |
| --- | --- |
| AsEnumerable | Returns the input typed as IEnumerable<T> |
| AsQuerable | Converts a (generic) iEnumerable to a (generic) IQuerable |
| Cast | Casts the elements of a collection to a specified type |
| OfType | Filters values, depending on their ability to be cast to a specified type |
| ToArray | Coverts the collection to an Array. This Method forces query execution |
| ToList | Converts the collection to a List<T>. This method forces query execution |
| ToDictionary | Puts the elements to a Dictionary<Tkey,Tvalue> based on key selector function. This method forces query execution |
| ToLookUp | Puts the elements into a LookUp<TKey, Tvalue> (a one-to-many dictionary) based on a key selector function. This method forces query execution. |

**Aggregation Operations**

* An aggregation operation computes a single value from the collection of values.
* Some of the aggregation functions are
  + Aggregate
  + Count / LongCount
  + Average
  + Max or MaxBy
  + Min or MinBy
  + Sum

**IQuerable v/s IEnumerable**

Both IQureable and IEnumerable used for data manipulation in LINQ from database and collections

|  |  |
| --- | --- |
| IQuerable | IEnumerable |
| It exists in System.Linq namespace | It exists in System.Collection namespace |
| IQuerable is suitable for querying data from out-memory (like sql server, remote machine etc) collection | IEnumerable is suitable for querying data in-memory collection like Array, List etc |
| It executes the “**select query**” on sql server **with all filters** | It executes the “**select query**” on sql server and **apply all filters on client-side** |
| IQuerable is benefited for LINQ to SQL queries | IEnumerable is benefited for LINQ to objects and LINQ to XML queries |

|  |
| --- |
| **Sameple Query**  IQueryable<Employee> empData = dbContext.Employees.Where(a=>a.JobTitle.Equals("Design Engineer"));  empData = empData.Take<Employee>(3);  **corresponding sql query**  xec sp\_executesql N'SELECT TOP(@\_\_p\_0) [e].[BusinessEntityID], [e].[BirthDate], [e].[CurrentFlag], [e].[Gender], [e].[HireDate], [e].[JobTitle], [e].[LoginID], [e].[MaritalStatus], [e].[ModifiedDate], [e].[NationalIDNumber], [e].[OrganizationLevel], [e].[rowguid], [e].[SalariedFlag], [e].[SickLeaveHours], [e].[VacationHours]  FROM [HumanResources].[Employee] AS [e]  WHERE [e].[JobTitle] = N''Design Engineer''',N'@\_\_p\_0 int',@\_\_p\_0=3  **IEnumerable**<Employee> empData = dbContext.Employees.Where(a=>a.JobTitle.Equals("Design Engineer"));  empData = empData.Take<Employee>(3);  **corresponding sql query**  SELECT [e].[BusinessEntityID], [e].[BirthDate], [e].[CurrentFlag], [e].[Gender], [e].[HireDate], [e].[JobTitle], [e].[LoginID], [e].[MaritalStatus], [e].[ModifiedDate], [e].[NationalIDNumber], [e].[OrganizationLevel], [e].[rowguid], [e].[SalariedFlag], [e].[SickLeaveHours], [e].[VacationHours]  FROM [HumanResources].[Employee] AS [e]  WHERE [e].[JobTitle] = N'Design Engineer'  Note: for IEnumerable only select query executes at sql server, other filter will apply on client side |