

Name :Mona Abdelmonem

Email: monasallam400@gmail.com

---

LINQ

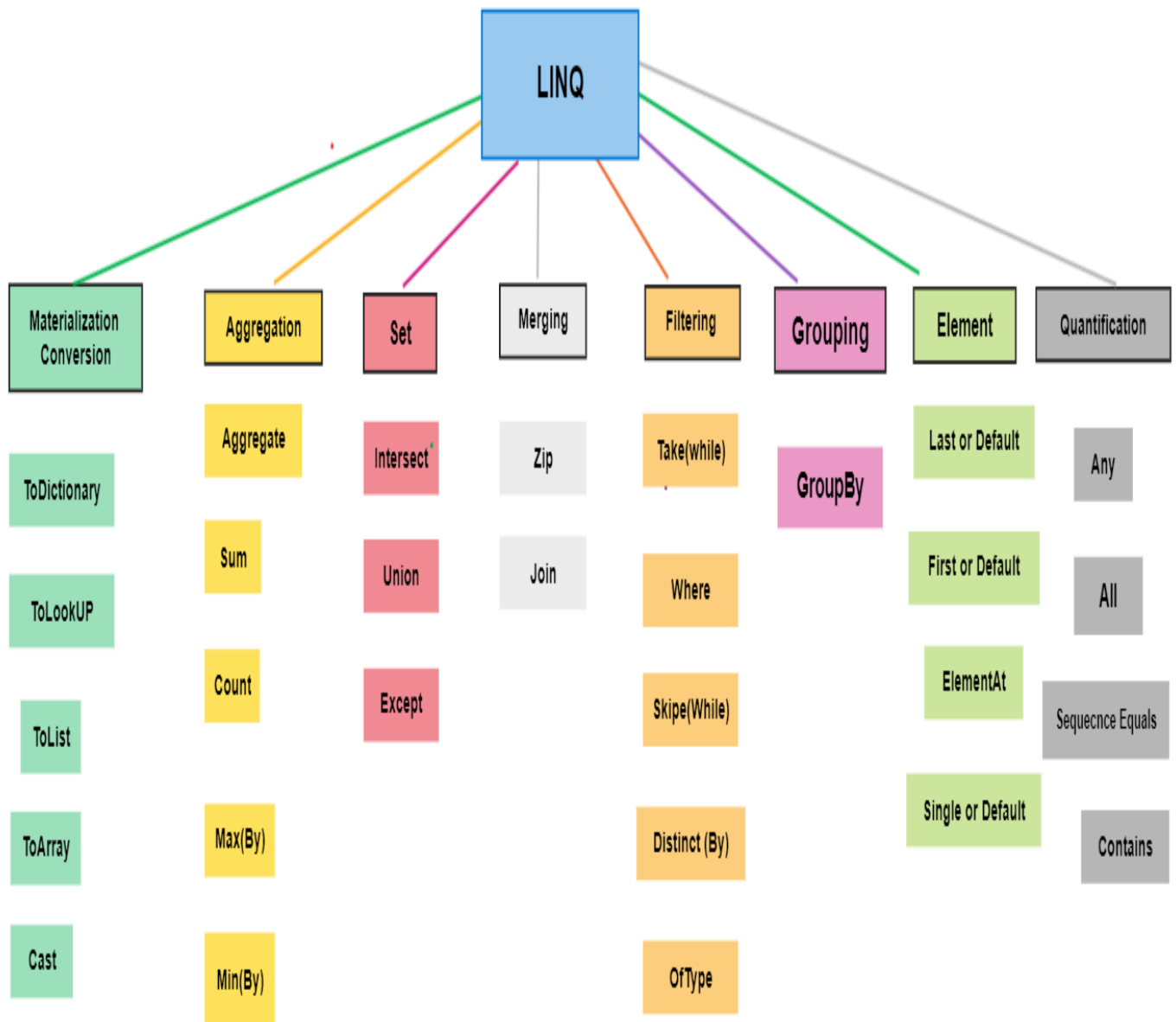
**Name:** Mona Abdelmonem

**Email:** monasallam400@gmail.com

**LinkedIn:** Mona Abdelmonem | LinkedIn

## LINQ:

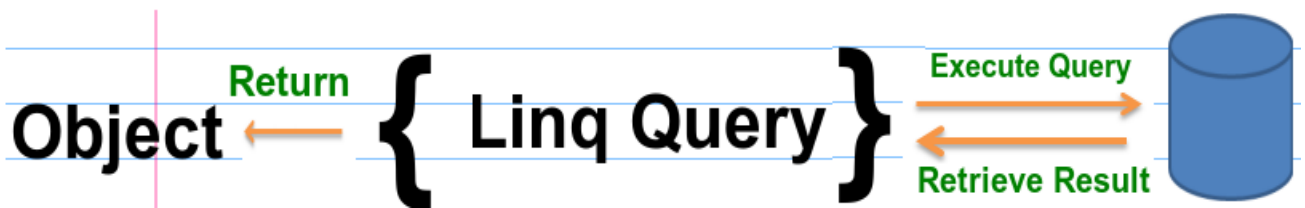
Language Integrated Query is the name for a set of technologies based on the integration of query capabilities directly into the C# language.



The target is to have a **uniform** and structured way to operate on enumerations.

LINQ queries return **always the result as new objects**. That ensures that the original enumeration will not be mutated.

LINQ queries return results as objects. It enables you to use object-oriented approach on the result set and not to worry about transforming different formats of results into objects.

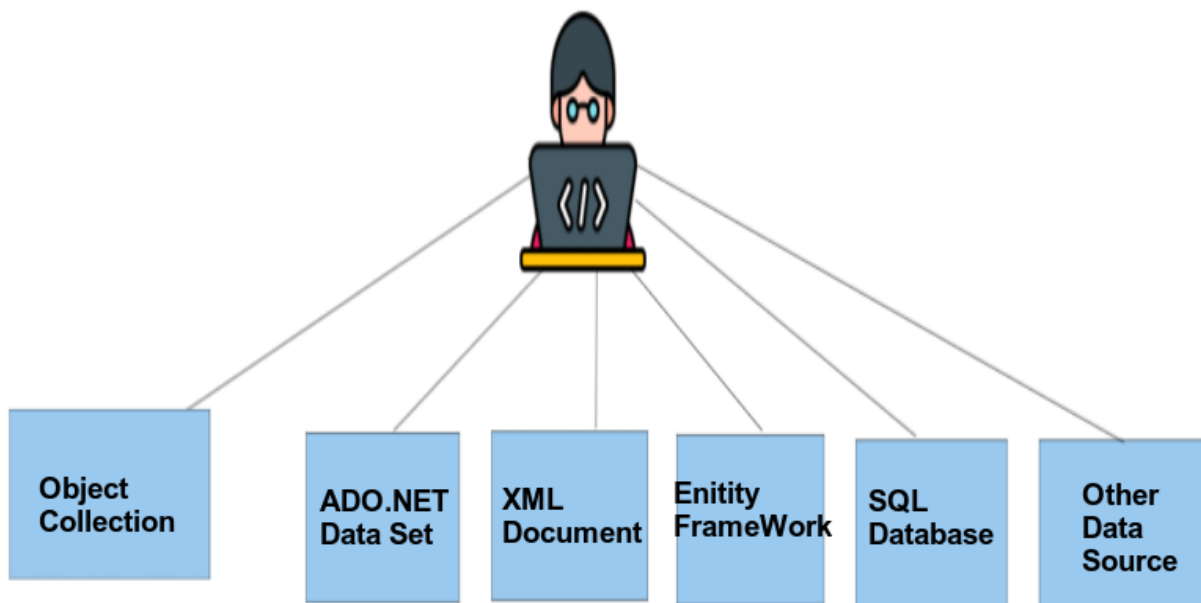


### Remember:

All LINQ queries return a new enumeration instead of deleting, updating or adding new items to the given one

A particular gateway for a particular data store (e.g. xml files, sql MySql, rdmbms) is called a **LINQ Provider**. It is realised by implementing the IQueryable Interface

For a developer who writes queries

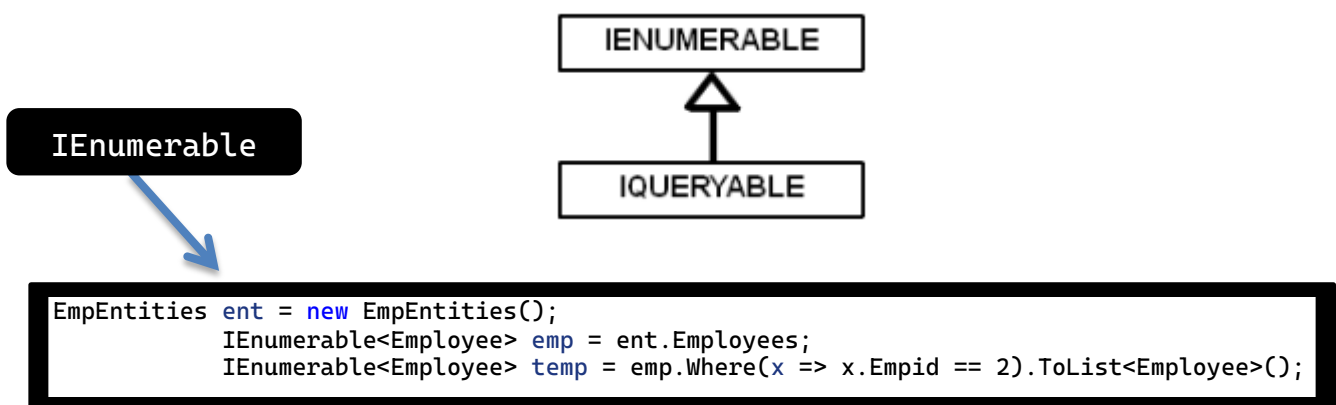


The basic type all LINQ queries operate on is **IEnumerable**.

## Why do we need **IEnumerable** and **IQueryable**?

Both IEnumerable and IQueryable are used for to hold the collection of a data and performing data manipulation operation for example filtering on the collection of data.

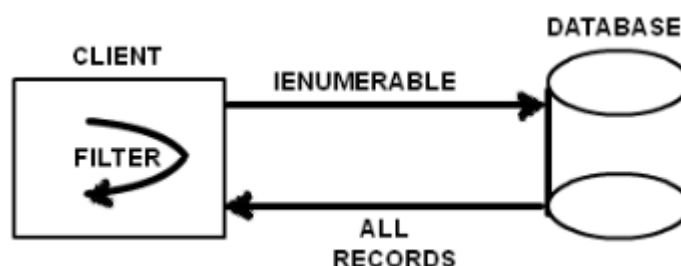
The first important point to remember is **IQueryable** interface **inherits** from **IEnumerable**, **so** whatever IEnumerable can do, IQueryable can also do



\*This where filter is executed on **the client side** where the IEnumerable code is.

\*In other words **all the data is fetched** from the database and then at the client its scans and gets the record with Empid is 2.

\*Filter at **Client Side**



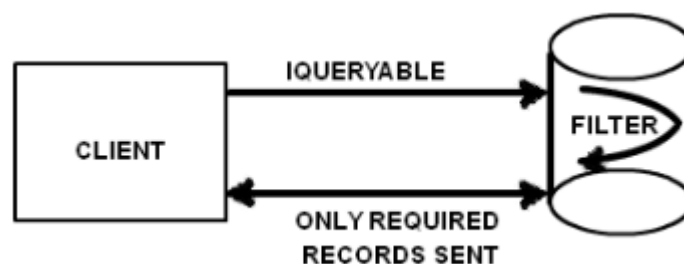
**IQueryable**

```
EmpEntities ent = new EmpEntities();  
IQueryable<Employee> emp = ent.Employees;  
IQueryable<Employee> temp = emp.Where(x => x.Empid == 2).ToList<Employee>();
```

\*It creates a SQL Query **at the server** side

\*In other words **only necessary** data is sent to the client side.

\*Filter at **Server Side**



why IQueryable is better choice for "**out-of-memory**" data?

why IEnumerable is better for "**in-memory**" data?

when it's "in-memory" it doesn't really matter. the data is already on the memory,

LINQ (Language Integrated Query) works as a **middle tier** between data store and the language environment.

## Comparison between **IEnumerable** And **IQueryable**

	<b>IEnumerable</b>	<b>IQueryable</b>
<b>Namespace</b>	System.Collection Namespace	System.Linq Namespace
<b>Derives form</b>	No base interfaces	Derives from IEnumerable
<b>Deferred Execution</b>	Supported	Supported
<b>Lazy Loading</b>	Not Supported	Supported
<b>How does it work</b>	While query date from database IEnumerable executes select query on server side load date in-memory  <b>On Client</b> side and then filter date .hence does <b>more work</b> and <b>become slow</b>	While query date from database IQueryable exectes select query  <b>On Server</b> side with all filters .hence does <b>less work</b> and <b>become fast</b>
<b>Suitable For</b>	Linq to Object  Linq to XML	Linq to SQL
<b>Custom Query</b>	Doesn't Support	Supported using createQuery and Execute method
<b>Extension method parameter</b>	Extention method supported in IEnumerable <b>takes functional</b> object	Extention method supported in IQueryable <b>takes expression</b> object i.e expression tree
<b>When to use</b>	When querying date <b>from in-memory</b> collection like List ,Array	When querying data <b>from out-memory</b> (like remote database ,server ) collection
<b>Best Use</b>	in-memory	out-of-memory /paging

```
//Data Source
string[] data = { "Mona", "Abdelmonem", "Soliman", "Sare"}

//Linq Query
var names=from d in data
           where d.Contains("M") || d.Contains("d")
           select d;

//Query execution
foreach (var item in names)
{
    Console.Write(item+" "); //Output:Mona Abdelmonem
}
```

## LINQ Method Syntax:

Method syntax (also known as **fluent syntax**)

Uses **Extension method** Included

OR

Enumerable

Queryable

**\*The compiler converts **query syntax** into **method syntax** at compile time**

The following is a sample LINQ method syntax query that returns a collection of strings which contains a Char "A".

```
IList<string> list = new List<string>
{
    "Mona Abdelmonem",
    "Alya Awny",
    "Habiba magdy",
    "Alaa Elomda",
    "Heba Mohamed",
    "Abdula Nagy"
};
var result = list.Where(x => x.Contains("A"));
foreach (var item in result)
{
    Console.WriteLine(item);
}
/*
Output:
Mona Abdelmonem
Alya Awny
Alaa Elomda
Abdula Nagy
*/
```

```
var result = list.Where(x => x.Contains("A"));
```

Extension Method

Lambda Expression

\*The real power of LINQ comes when you combine multiple operations are used in one statement .

```
IEnumerable<BlogPost> allBlogPosts = await GetAllBlogPosts();  
var publishedBlogPosts = allBlogPosts  
    .Where(bp => bp.IsPublished)  
    .OrderByDescending(bp => bp.PublishDate)  
    .Skip(pageSize * (page - 1))  
    .Take(pageSize)  
    .ToList();
```

## Filtering

- where
- Take
- Skip
- OfType
- Distinct

-use LINQ to filter the enumeration based on the given operation.

## Where Clause?

- The Where clause is a query operator that filters a sequence based on a specified condition.
- It returns a new sequence that contains only the elements that satisfy the condition





```
var res = result.Where(x=>x.Shape=='🐼');
```



- The method accepts a Predicate.
- That means we define a filter function which then gets applied object by object.
- If the filter evaluates to true, the element will be returned in the new enumeration.

### Query Syntax:

```
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
list.Add(4);
var res=from li in list
        where ( li % 2 == 0)
        select li;
//Result: [2,4]
```

### Lambda Syntax:

```
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
list.Add(4);
var res=list.Where(x=>x%2==0);
//Result: [2,4]
```

### Take

- Take allows us to "take" the given amount of elements.  
If we have less elements in the array
- Take() will only return the remaining objects.



```
var res = list.Take(2);
```



### Lambda Syntax:

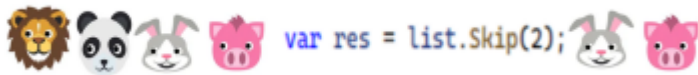
```
var list = new List<int>();
list.Add(1);
list.Add(5);
list.Add(3);
list.Add(3);
list.Add(4);
var res = list.Take(2);
//Result: [1,5]
```

### Query Syntax:

```
list.Add(1);
list.Add(5);
list.Add(3);
list.Add(4);
var res = (from li in list
           select li).Take(2);
////Result: [1,5]
```

## Skip

- With Skip we "skip" the given amount of elements
- If we skip more elements than our list holds, we get an empty enumeration back.



### Note:

Take and Skip together can be very powerful for stuff like pagination.

## Distinct

- returns a new enumerable where all duplicates are removed .
- Distinct the level of the object



```
var list = new List<int>();
list.Add(1);
list.Add(1);
list.Add(2);
var uniqueElements = list.Distinct();
foreach (var item in uniqueElements)
{
    Console.Write(item+" ");
}
//Output
// [ 1, 2 ]
```

## DistinctBy

works similar to Distinct but instead of the level of the object itself we can define a projection to a property where we want to have a distinct result set.

--DistinctBy the level of the property



```
var people = new List<Person>
{
    new Person("Mona", 22),
    new Person("Ali", 29),
    new Person("Mona", 22),
    new Person("Abdula", 29),
    new Person("Mona", 22)
};
var uniqueAgedPeople = people.DistinctBy(p => p.Age); //Distinct By Age
foreach (var item in uniqueAgedPeople)
{
    Console.WriteLine(item.Name + " " + item.Age);
}
/*
Output:
Mona 22
Ali 29
*/
```

## OfType

OfType checks every element in the enumeration if it is of a given type

-That helps especially if we have untyped arrays (object)

-we want a special subclass of the given enumeration.



```
var fruits = new List<Fruit>
{
    new Banana(),
    new Apple()
};

var apples = fruits.OfType<Apple>();
```

# Projection

-Select

-Select Many

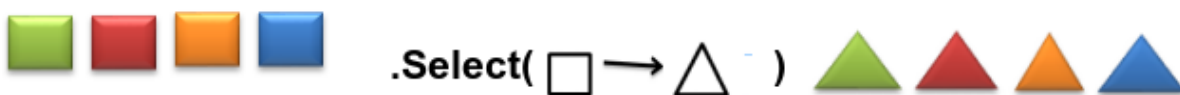
\*Projection describes the transformation of an object into a new form.

\*By using projections, you can create a new type which is built from your original type.

## Select

-we map from our a given type to a desired type.

-The result set has the same amount of items as the source set.



## SelectMany

-is used to flatten lists.

-If you have a list inside a list we can use it to flatten this into a one dimensional representation.



# Aggregation

-Count

- Aggregate

-Max(By)

Aggregation describes the process of reducing the whole enumeration to a single value.

## Count

we count elements by a given function. If the function evaluates to true, we increase the counter by one.

## Aggregate

Aggregate, also known as reduce, aggregates/reduces all elements into a scalar value.

```
var numbers = new[] { 1, 2, 3 };

var sum = numbers.Aggregate(0, (curr, next) => curr + next);
//Output:6

var sumLinq = numbers.Sum();
// Output:6
```

## Max(By)

-Max(By) retrieves the biggest element.

-If Max or MaxBy is presented an empty enumeration it will also return an empty enumeration.

```
var max = new[] { 1, 2, 3 }.Max();

var people = new[]
{
    new Person(15, "Mona"),
    new Person(7, "Abdelmonem")
};

var oldest = people.MaxBy(p => p.id);
Console.WriteLine(oldest.name);
//Output: Mona
```

# Quantification

Those operations want to measure the quantity of something.

-Any

-All

-SequenceEquals

## Any

-Any checks if at least one element satisfies your condition. If so, it returns true.

-If there is no element that meets the condition, then it returns false.






-It returns false if the given enumeration is empty

    → .Any(  )    // True

```
var max = new[] { 1, 2, 3 }.Max();  
var people = new[]  
{  
    new Person(15, "Mona"),  
    new Person(7, "Abdelmonem")  
};  
var oldest = people.Any(x => x.id > 9);  
Console.WriteLine(oldest);  
//Output: True  
var oldest2 = people.Any(x => x.id < 5);  
Console.WriteLine(oldest2);  
//Output: False
```

## All






-checks if All of your elements in the list satisfy a certain condition. If so returns true, otherwise false.

    → .All (  )    // False

```
var max = new[] { 1, 2, 3 }.Max();  
var people = new[]  
{  
    new Person(15, "Mona"),  
    new Person(7, "Abdelmonem")  
};  
var oldest = people.All(x => x.id > 9);  
Console.WriteLine(oldest);  
//Output: False  
var oldest2 = people.All(x => x.id == 7);  
Console.WriteLine(oldest2);  
//Output: False
```

## SequenceEquals

- SequenceEquals checks if two sequences are equal.
- It uses the default equality comparer
- Two empty lists are also equal
- There is an optional second parameter which allows to pass in an IEqualityComparer
- That is useful if you don't have control over the type

    .SequenceEquals(     ) // True

```
var numbers = new[] { 1, 2, 3, 4 };  
var numbers2 = new[] { 1, 2, 4, 3 };  
  
var equal = numbers.SequenceEqual(numbers2);  
// Output: false
```

## Merging

-Join

-ZIP

## Join

responsible of merging two or more enumerations into one object.

Join works similar to a SQL Left-Join

```
var res = from e1 in emp1
           join e2 in emp2
             on e1.emp_id equals e2.emp_id
           select new
           {
               Emp_Name = e1.emp_name,
               Emp_Salary = e2.emp_salary
           };
```

## Zip

-With Zip we "merge" two lists by a given merge function.

-We merge objects together until we run out of objects on either of the lanes.

-The first list contains 9 element and the second list contain 3 element

So the result will contain the 3 element



```
var letters = new[] { 1,2,4,5,6,7,8,12,20 };
var numbers = new[] { 1, 2, 3 };

var merged = letters.Zip(numbers, (x,y) => x*y);
foreach (var item in merged)
{
    Console.WriteLine(item+" ");
}
//Output: 1 4 12
```



# Element

-First(or Default)

-Second(or Default)

-ElementAt()

-Last(or Default)

how to retrieve a specific item from the enumeration

## First

- returns the first occurrence of an enumeration

-If no element is found, it throws an exception.

```
var people = new[]
{
    new Person(30, "Mona"),
    new Person (15, "Abdelmone"),
    new Person(30, "Noor")
};
var res =people.First(x=>x.Age==30 );
Console.WriteLine(res.Age+" "+res.Name);
//Output: 30 Mona
```

## Single

Single does not return immediately after the first occurrence

-The difference to First is that Single ensures there is not a second item of the given type

-Single has to go through the whole enumeration (worst case)

.if it can find another item it throws an exception.

.If no element is found, it throws an exception.

```
var people = new[]
{
    new Person(30, "Mona"),
    new Person (15, "Abdelmone"),
    new Person(30, "Noor")
};

var res =people.Single(x=>x.Name=="Mona" );

Console.WriteLine(res.Age+" "+res.Name);
//Output: 30 Mona

var res2 = people.Single(x => x.Age == 30);
Console.WriteLine(res2.Age + " " + res2.Name);
//Output:
//This throws an exception as there are
// Sequence contains more than one matching element
```

## FirstOrDefault / SingleOrDefault

If no element is found in the given enumeration it returns it the default

- for reference types null

- for value types the given default like 0 for an integer

. Since .NET6 we can pass in what "default" means to us.

```
var people = new[]
{
    new Person(30, "Mona"),
    new Person (15, "Abdelmone"),
    new Person(30, "Noor")
};

var res =people.FirstOrDefault(x=>x.Name=="Ali");

Console.WriteLine(res);
//Output :Null

var res2 = people.SingleOrDefault(x => x.Age == 50, new Person(0, "Not Found"));
Console.WriteLine(res2.Age + " " + res2.Name);
//Pass Default Parameter
//Output:0 Not Found
```

# Materialisation / Conversion

-List

-ToArray

-ToLookup

-ToDictionary

-Cast

A lookup is defined that we have a key which can point to list of objects (1 to n relation).

The first argument takes the "key"-selector.

The second selector is the "value".

A LookUp-object is immutable. You can't add elements afterwards.

```
var products = new[]
{
    new Product("Smartphone", "Electronic"),
    new Product("PC", "Electronic"),
    new Product("Apple", "Fruit")
};

var lookup = products.ToLookup(k => k.Category, elem => elem);
```

## ToDictionary

-ToDictionary works similar to ToLookup with a key difference.

-The ToDictionary method only allows 1 to 1 relations

-If two items share the same key, it will result in an exception that the key is already present.

-Also the dictionary can be mutated afterwards

```
var products = new[]
{
    new Product(1, "Smartphone"),
    new Product(2, "PC"),
    new Product(3, "Apple")
};

var idToProductMapping = products.ToDictionary(k => k.Id, elem =>elem);
// Product { Id: 1, Name: "Smartphone" }
var itemWithId1 = idToProductMapping[1];
```

## ToList / ToArray

- objects of the type Enumerable are not evaluated directly but only when they are materialised.

```
var list = new List<int>();
list.Add(1);
list.Add(2);
var evenNumbers = list.Where(n => n % 2 == 0).ToList();
list.Add(4);
list.Add(8);
list.Add(10);
// Even numbers in list: 1
// as we materialised the list
Console.WriteLine($"Even numbers in list: {evenNumbers.Count()}");
```

**Notes :** If we don't write the .ToList() //Output!!!!

```
var list = new List<int>();
list.Add(1);
list.Add(2);
var evenNumbers = list.Where(n => n % 2 == 0);
list.Add(4);
list.Add(8);
list.Add(10);
// Even numbers in list: 4
Console.WriteLine($"Even numbers in list: {evenNumbers.Count()}");
```

## Grouping

-GroupBy

- GroupBy groups the enumeration by a given projection / key
- All elements which share this exact key get grouped together
- It is almost identical to ToLookup with a very big difference
- Calling ToLookup means "I want a cache of the entire thing right now organised by group."

```
var products = new[]  
{  
    new Product("Smartphone", "Electronic"),  
    new Product("PC", "Electronic"),  
    new Product("Apple", "Fruit")  
};  
  
var lookup = products.GroupBy(k => k.Category, elem => elem);
```

## Set

-Intersect

-Union

-Except

which behave like sets. Sets are specially in the sense that they only hold distinct (disjoint) objects in them

-Union

The union of two lists will result in every distinct element which is in both of your lists  
It behaves like a set, so duplicated items are removed.



```
var numbers1 = new[] { 1, 1, 2 };  
var numbers2 = new[] { 2, 3, 4 };  
  
var result = numbers1.Union(numbers2);  
  
// Output: [ 1, 2, 3, 4 ]
```

## Intersect :

Intersect works similar to Union but now we check which elements are present in list A AND list B

-Only unique items are in the new list. Duplicates are automatically removed.

```
var numbers1 = new[] { 1, 1, 2 };
var numbers2 = new[] { 2, 3, 4 };

var result = numbers1.Intersect(numbers2);

// Output: [ 2 ]
```

## The different between **Deferred Execution** & **Immediate Execution**

### 1. Deferred or Lazy Operators:

These query operators are used for deferred execution. **For example** – select, SelectMany, where, Take, Skip, etc. are belongs to Deferred or Lazy Operators category.

### 2. Immediate or Greedy Operators:

These query operators are used for immediate execution. **For Example** – count, average, min, max, First, Last, ToArray, ToList, etc. are belongs to the Immediate or Greedy Operators category.

## Deferred Execution:

it doesn't execute by itself. It executes only when we access the query results.

So, here the execution of

the query is deferred until the query variable is iterated over using for each loop.

```
List<Employee> listEmployees = new List<Employee>
{
    new Employee { ID= 1001, Name = "Priyanka", Salary = 80000 },
    new Employee { ID= 1002, Name = "Anurag", Salary = 90000 },
    new Employee { ID= 1003, Name = "Preety", Salary = 80000 }
};
IEnumerable<Employee> result = from emp in listEmployees
                               where emp.Salary == 80000
                               select emp; //Deferred Execution

listEmployees.Add(new Employee { ID = 1004, Name = "Santosh", Salary = 80000 });
foreach (Employee emp in result)
{
    Console.WriteLine($" {emp.ID} {emp.Name} {emp.Salary}");
}
```

## Output:

```
1001 Priyanka 80000
1003 Preety 80000
1004 Santosh 80000
```

## Immediate Execution:

it forces the query to execute and gets the result immediately.

Let us see an example for a better understanding.

```
List<Employee> listEmployees = new List<Employee>
{
    new Employee { ID= 1001, Name = "Priyanka", Salary = 80000 },
    new Employee { ID= 1002, Name = "Anurag", Salary = 90000 },
    new Employee { ID= 1003, Name = "Preety", Salary = 80000 }
};
IEnumerable<Employee> result = (from emp in listEmployees
                                where emp.Salary == 80000
                                select emp).ToList();//Immediate Execution
listEmployees.Add(new Employee { ID = 1004, Name = "Santosh", Salary = 80000 });
foreach (Employee emp in result)
{
    Console.WriteLine($" {emp.ID} {emp.Name} {emp.Salary}");
}
Console.ReadKey();
```

## Output:

```
1001 Priyanka 80000
1003 Preety 80000
```

# Exercises

## Numbers from range Easy

Given an array of integers, write a query that returns list of numbers greater than 30 and less than 100.

### Expected input and output

[67, 92, 153, 15] → 67, 92

[Click here to see example solution](#)

## Minimum length Easy

Write a query that returns words at least 5 characters long and make them uppercase.

### Expected input and output

"computer", "usb" → "COMPUTER"

[Click here to see example solution](#)

## Select words Easy

Write a query that returns words starting with letter 'a' and ending with letter 'm'.

### Expected input and output

"mum", "amsterdam", "bloom" → "amsterdam"

[Click here to see example solution](#)

## Top 5 numbers Easy

Write a query that returns top 5 numbers from the list of integers in descending order.

### Expected input and output

[78, -9, 0, 23, 54, 21, 7, 86] → 86 78 54 23 21

[Click here to see example solution](#)



---

### Square greater than 20 Easy

Write a query that returns list of numbers and their squares only if square is greater than 20

**Expected input and output**

[7, 2, 30] → 7 - 49, 30 - 900

[Click here to see example solution](#)

---

### Replace substring Easy

Write a query that replaces 'ea' substring with astersik (\*) in given list of words.

**Expected input and output**

"learn", "current", "deal" → "l\*rn", "current", "d\*l"

[Click here to see example solution](#)

---

### Last word containing letter Easy

Given a non-empty list of words, sort it alphabetically and return a word that contains letter 'e'.

**Expected input and output**

["plane", "ferry", "car", "bike"]→ "plane"

[Click here to see example solution](#)

---

### Shuffle an array Medium

Write a query that shuffles sorted array.

**Expected input and output**

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] → [4, 9, 3, 5, 2, 10, 1, 6, 8, 7]

[38, 24, 8, 0, -1, -17, -33, -100] → [0, -100, -17, 38, 8, -1, 24, -33,]

[Click here to see example solution](#)

---

### Decrypt number Medium

Given a non-empty string consisting only of special chars

{ '(', ')', '!', '@', '#', '\$', '%', '^', '&', '\*', '(', ')' } return a number (as a string) where each digit corresponds to given special char on the keyboard

**Expected input and output**

"() ) (" → "9009"

"\*\$ (#&" → "84937"

"!!!!!!!!!!!!" → "1111111111"

[Click here to see example solution](#)

---

### Most frequent character Medium

Write a query that returns most frequent character in string. Assume that there is only one such character.

**Expected input and output**

"panda" → 'a' "n093nfv034nie9" → 'n'

[Click here to see example solution](#)

---

### Unique values Medium

Given a non-empty list of strings, return a list that contains only unique (non-duplicate) strings.

**Expected input and output**

["abc", "xyz", "klm", "xyz", "abc", "abc", "rst"] → ["klm", "rst"]

[Click here to see example solution](#)

---

### Uppercase only Medium

Write a query that returns only uppercase words from string.

**Expected input and output**

"DDD example CQRS Event Sourcing" → DDD, CQRS

[Click here to see example solution](#)

Name :Mona Abdelmonem

Email: monasallam400@gmail.com

---

### Arrays dot product Medium

Write a query that returns dot product of two arrays.

#### Expected input and output

`[1, 2, 3], [4, 5, 6] → 32 [7, -9, 3, -5], [9, 1, 0, -4] → 74`

[Click here to see example solution](#)

---

### Frequency of letters Medium

Write a query that returns letters and their frequencies in the string.

#### Expected input and output

`"gamma" → "Letter g occurs 1 time(s), Letter a occurs 2 time(s), Letter m occurs 2 time(s)"`

[Click here to see example solution](#)

---

### Important Links

- [Mona400/LINQ: Exercises & solutions \(github.com\)](#)
- [Language-Integrated Query \(LINQ\) \(C#\) | Microsoft Learn](#)
- [LINQ Tutorial For Beginners and Professionals - Dot Net Tutorials](#)
- [C# programming exercises - examples with solutions \(csharpexercises.com\)](#)
- [All C# Linq Program With Examples | Techstudy](#)

Name :Mona Abdelmonem

Email: monasallam400@gmail.com

---

# THANK YOU

Email: [monasallam400@gmail.com](mailto:monasallam400@gmail.com)

LinkedIn: [Mona Abdelmonem | LinkedIn](#)