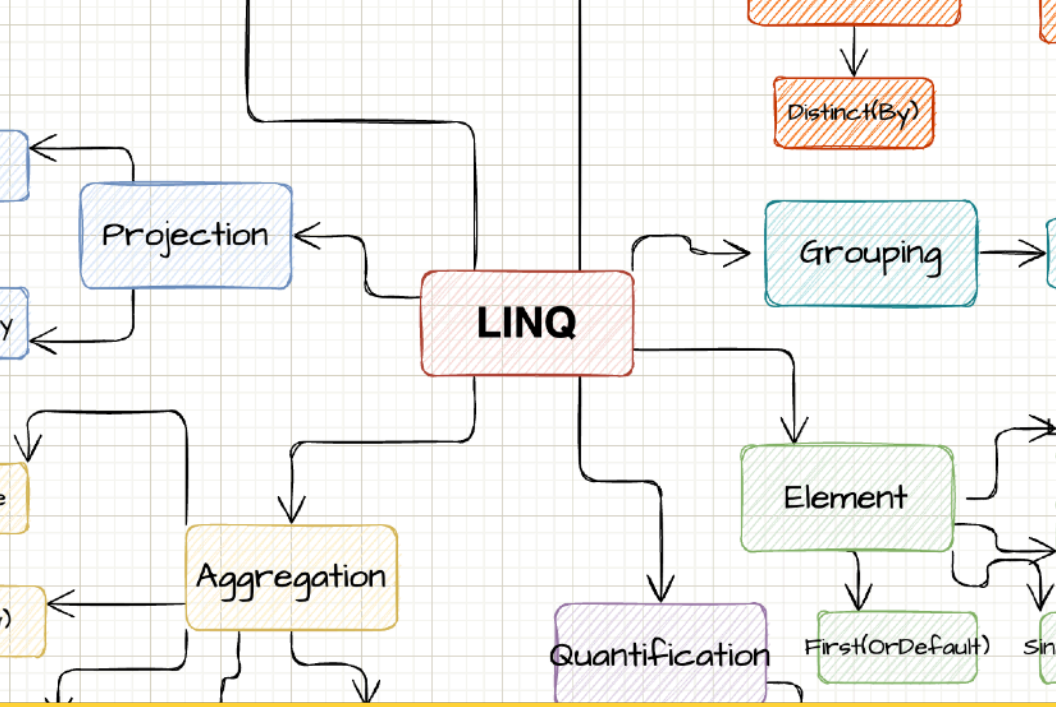


LINQ

Explained with sketches

By Steven Giesel

Prologue

This little piece was created with the ulterior motive of giving beginners a visual and simple introduction to LINQ. According to the motto: "Pictures say more than a thousand words".

What is the goal of this little book? It should enable you to use the right LINQ queries in the right situation. I will start each time with a small drawing, which will be completed by a small explanation including a code example. I'm going to go less into things like "LINQ-To-Object". Also I will only very briefly touch things like **IQueryable** and **IEnumerable**.

Table of contents

What is LINQ?

LINQ short for “Language-Integrated Query” is the name for a set of technologies based on the integration of query capabilities directly into the C# language.[1]

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. This is very important to remember. All LINQ queries return a new enumeration instead of deleting, updating or adding new items to the given one.

Furthermore there are ways to transform LINQ queries to SQL syntax or use LINQ to go through a XML document. The basic type **all** LINQ queries operate on is **IEnumerable**.

---

[1] Definition: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

IEnumerable

The basic type all LINQ queries operate on is IEnumerable. Without going into too much detail, it is crucial to understand that **IEnumerable** does not represent a "materialised" list. We call this "lazy evaluation". That means at the the time of calling LINQ queries we don't get the actual results. Only when we enumerate through the enumeration or call operations like **ToList**, or **Count** we really "create" / "materialise" the object.

Now you will see a small snippet. Don't worry if you don't understand this now. Take it as a motivation to fully understand this after you read the small book:

```
var list = new List<int>();
list.Add(1);
list.Add(2);

var evenNumbers = list.Where(n => n % 2 == 0);

list.Add(4);
Console.WriteLine($"Even numbers in list:
{evenNumbers.Count()}");
```

We create the enumeration after the list holds 2 elements (1 and 2). Afterwards we add another number to the list itself. So how many even numbers do we have in the enumeration? The answer is: **2**. The reason is that we materialise on **Count** and not at the moment of creating the enumeration in the first place. So when we call **Count** we have two elements, which are even numbers (2 and 4). Always keep that in mind.

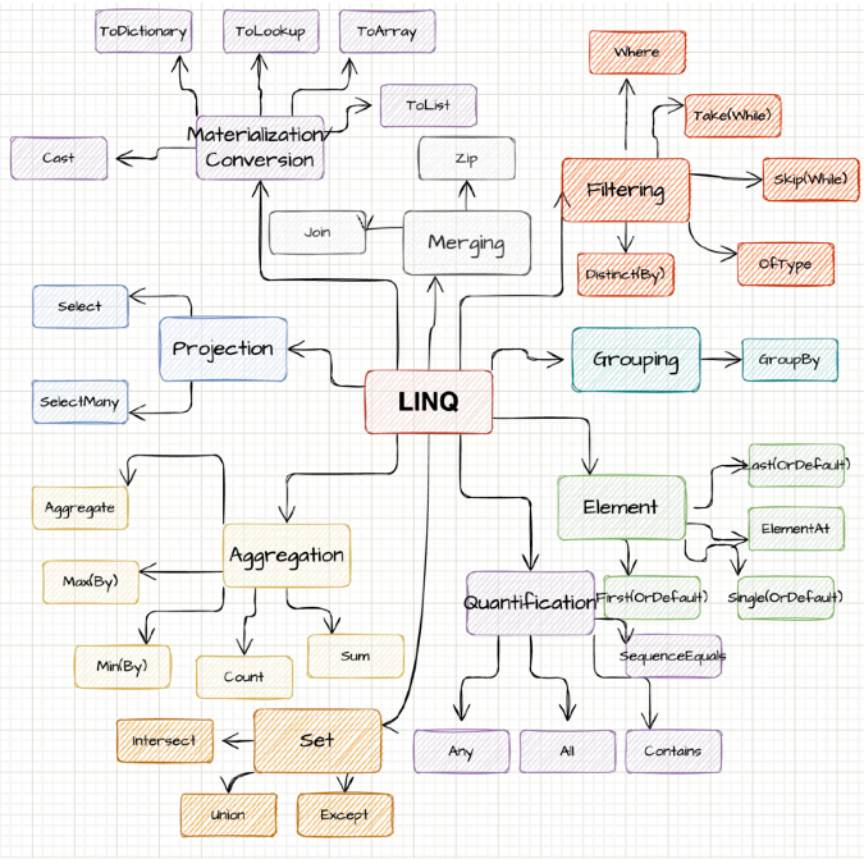
Another type always associated with LINQ is IQueryable. IQueryable is basically IEnumerable plus more and is exactly that “plus more” part which makes it so unique. For that I will just list the highlights here and reference to my blog post: IEnumerable vs IQueryable - What's the difference which will go into greater detail.

- Both IEnumerable and IQueryable are forward collections - they don't get materialised right away
- Querying data from the database IEnumerable will load the data into memory in filter afterwards on the client
- Querying data from the database IQueryable will filter first and afterwards send the filtered data to the client
- IQueryable is suitable for querying data from out-memory
- There can be scenarios where the underlying query provider can't translate your expression to something meaningful then you have to switch to IEnumerable

Mindmap

LINQ has many many operations in its toolset for you. So we can group them in different categories. The next picture will show you a rough overview so that you can get a mental picture. I would also advice to come back to that image time and time again to see where you at.

The upcoming chapters are organised by exactly those categories.



The real power of LINQ comes when you combine multiple operations. After the explanation of the LINQ operators you will find some real world samples where multiple LINQ 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

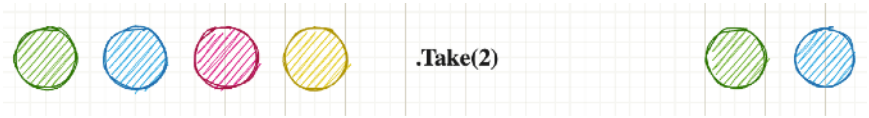
The following chapter describes how one can use LINQ to filter the enumeration based on the given operation.

Where

With **Where** we can filter a given list based on our condition. 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.

```
var list = new List<int>();
list.Add(1);
list.Add(2);

// Get even numbers
// Result: [ 2 ]
var evenNumbers = list.Where(n => n % 2 == 0);
```

Take



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

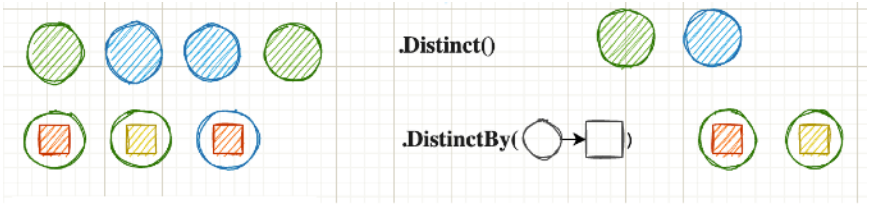
```
var list = new List<int>();
list.Add(1);
list.Add(2);

// Result: [ 1 ]
var takeOne = list.Take(1);

// Result: [ 1, 2 ]
var takeOneHundred = list.Take(100);
```

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

```
var list = new List<int>();
list.Add(1);
list.Add(1);
list.Add(2);

// [ 1, 2 ]
var uniqueElements = list.Distinct();


var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
```

Distinct(By)

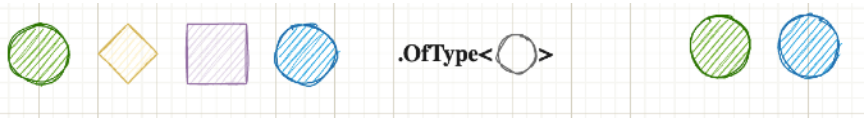


Distinct returns a new enumerable where all duplicates are removed, kind of like a Set. Be careful that for reference type the default is to check for equality of references, which can lead to false results. The result set can be the same or smaller.

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.

```
var people = new List<Person>
{
    new Person("Steven", 31),
    new Person("Katarina", 29),
    new Person("Nils", 31)
};

// [
//     Person { Name = Steven, Age = 31 },
//     Person { Name = Katarina, Age = 29 }
// ]
var uniqueAgedPeople = people.DistinctBy(p => p.Age);

record Person(string Name, int Age);
```

OfType



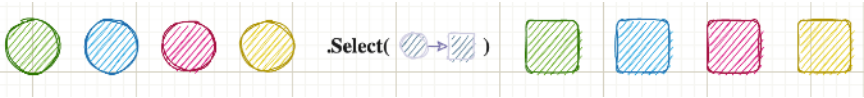
OfType checks every element in the enumeration if it is of a given type (also inherited types count as that given type) and returns them in a new enumeration. That helps especially if we have untyped arrays (object) or we want a special subclass of the given enumeration.

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

// [
//     Apple { }
// ]
var apples = fruits.OfType<Apple>();

record Fruit;
record Banana : Fruit;
record Apple : Fruit;
```

Projection

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



With **Select** we create a projection from one item to another. Simply speaking we map from our a given type to a desired type. The result set has the same amount of items as the source set.

```
var objects = new List<SourceObject>
{
    new SourceObject(1),
    new SourceObject(2),
};

// [
//     TargetObject { NumberAsString: "1" },
//     TargetObject { NumberAsString: "2" },
// ]
var targetObjects = objects.Select(o => new
TargetObject(o.ToString()));

record SourceObject(int Number);
record TargetObject(string NumberAsString);
```

SelectMany

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.

```
var recipes = new List<Recipe>
{
    new Recipe("Pizza", new() { "Tomato Sauce, Basil" }),
    new Recipe("Hot Water", new() { "Water" }),
};

// [
//     "Tomato Sauce", "Basil", "Water"
// ]
var allIngredients = recipes.SelectMany(r => r.Ingredients);

record Recipe(string Name, List<string> Ingredients);
```

Aggregation

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

Count

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

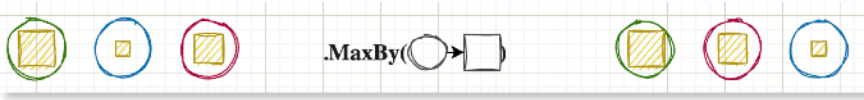
```
var names = new[] { "Steven", "Marie", "Steven" };

// 2
var stevens = names.Count(n => n == "Steven");
```

Aggregate

Aggregate, also known as **reduce**, aggregates/reduces **all** elements into a scalar value. A prime example is the sum of a list. We start with 0 and add each element on top until we enumerated through our enumeration. Aggregates first parameter is the start value. An empty enumeration will result in returning your start value.

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

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

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

Max(By)



Max(By) retrieves the biggest element. This also can be represented by an aggregate function. If **Max** or **MaxBy** is presented an empty enumeration it will throw an exception, that the sequence contains no element.

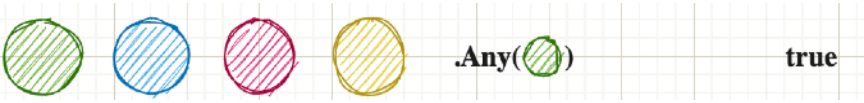
```
// 3
var max = new[] { 1, 2, 3 }.Max();

var people = new[]
{
    new Person("Steven", 31),
    new Person("Jean", 22)
};

// Person { Name: Steven, Age: 31 }
var oldest = people.MaxBy(p => p.Age);

record Person(string Name, int Age);
```

Of course **Min(By)** works similar. The difference is of course that the smallest value is retrieved instead of the biggest.

Quantification

This chapter looks into quantification of elements. Those operations want to measure the quantity of something.

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. Any also immediately stops processing once it founds one element. It returns **false** if the given enumeration is empty.

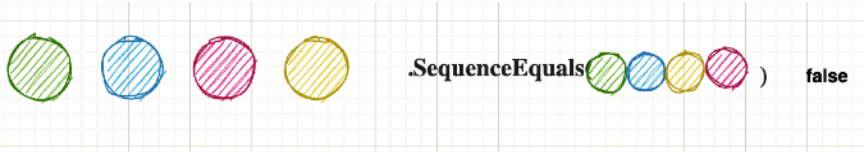
```
var fruits = new[]
{
    new Fruit("Banana", 89),
    new Fruit("Apple", 51),
};

// true
var hasDenseFood = fruits.Any(f => f.CaloriesPer100Gramm > 80);

record Fruit(string Name, int CaloriesPer100Gramm);
```

All

As the name implies checks if **All** of your elements in the list satisfy a certain condition. If so returns true, otherwise false. If **All** finds an element which does not satisfy the condition it immediately stops processing and returns **false**.

```
var fruits = new[]
{
    new Fruit("Banana", 89),
    new Fruit("Apple", 51),
};

// false
var hasDenseFood = fruits.All(f => f.CaloriesPer100Gramm > 80);

record Fruit(string Name, int CaloriesPer100Gramm);
```

SequenceEquals



SequenceEquals checks if two sequences are equal. Equal means they have the same amount of entries inside the enumeration as well as all elements are equal. It uses the default equality comparer. Two empty lists are also equal.

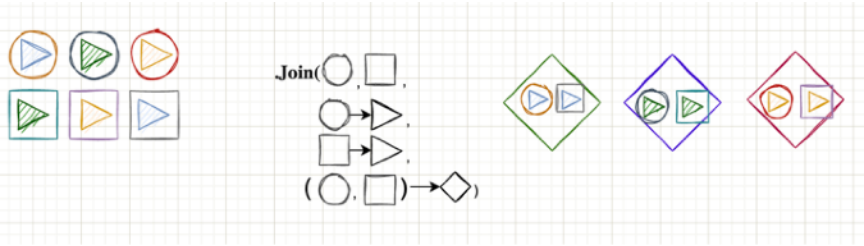
There is an optional second parameter which allows to pass in an IEqualityComparer. That is useful if you don't have control over the type and therefore can't override Equals. By default reference types are compared by their references against each other, which is not always what you want.

```
var numbers = new[] { 1, 2, 3, 4 };
var moreNumbers = new[] { 1, 2, 4, 3 };

// false
var equal = numbers.SequenceEqual(moreNumbers);
```

Merging

This chapter looks into operations which are responsible of merging two or more enumerations into one object.

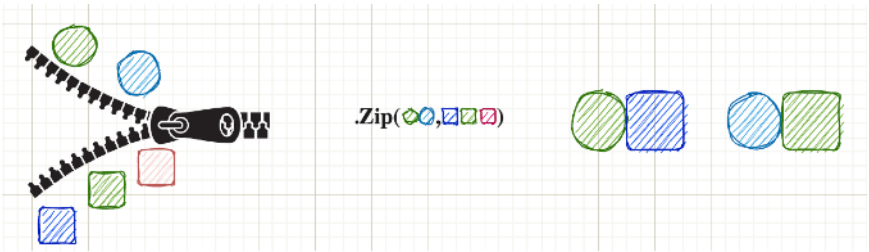
Join



Join works similar to a **SQL** Inner-Join. We have two sets we want to join. The next two arguments are the "key" selectors of each list. What Join basically does is it takes every element in list A and compares it with the given "key-selector" against the key-selector of list b. If it matches we can create a new object C, which can consist out of those two elements.

```
var fruits = new[]
{
    new Fruit(1, "Banana", 89),
    new Fruit(2, "Apple", 51),
};

var classification = new[]
{
    new FruitClassification(1, "Magnesium-rich")
};

// { Name = Banana, Classification = Magnesium-rich }

var fruitWithClassification = fruits.Join(
    classification,
    f => f.FruitId, c => c.FruitId,
    (f, c) => new { f.Name, Classification = c.Classification });

foreach(var t in fruitWithClassification) Console.Write(t);

record Fruit(int FruitId, string Name, int CaloriesPer100Gramm);
record FruitClassification(int FruitId, string Classification);
```

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. As seen in the example: The first lane has 2 elements, the second has 3. Therefore the result set contains only 2 elements.

```
var letters = new[] { "A", "B", "C", "D", "E" };
var numbers = new[] { 1, 2, 3 };
// [ "A1", "B2", "C3" ]
var merged = letters.Zip(numbers, (l, n) => l + n);
```

Element

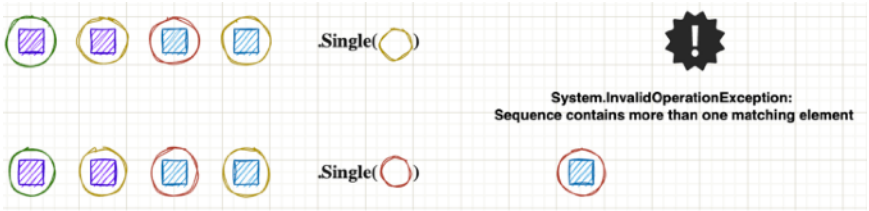
This chapter looks closer how to retrieve a specific item from the enumeration.

First

First returns the first occurrence of an enumeration. Even if there are elements later it always returns immediately after the first found item. If no element is found, it throws an **exception**.

```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// Person { Name: Steven, Age: 31 }
var firstOver30 = people.First(p => p.Age > 30);

record Person(string Name, int Age);
```

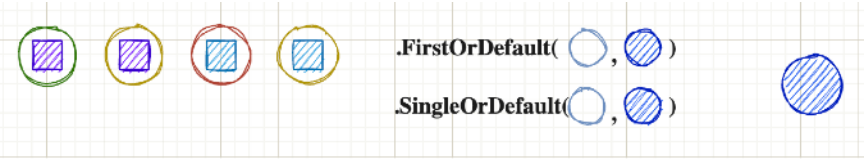
Single



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 / predicate. Therefore **Single** has to go through the whole enumeration (worst case) if it can find another item. If so, it throws an exception. If no element is found, it throws an **exception**.

```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// Person { Name: Steven, Age: 31 }
var steven = people.Single(p => p.Name == "Steven");

// This throws an exception as there are
// multiple people above 30
var above30 = people.Single(p => p.Age > 30);

record Person(string Name, int Age);
```

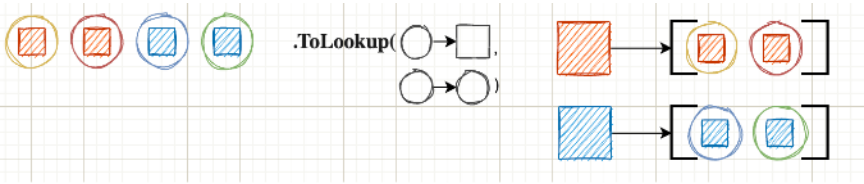
FirstOrDefault / SingleOrDefault



If no element is found in the given enumeration it returns it the default (for reference types **null** and for value types the given default like 0 for an integer). Since .NET6 we can pass in what "default" means to us. Therefore we can have non-nullable reference types if we wish or any given number / float / string.

```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// null, as the default of a reference type is null
var steven = people.FirstOrDefault(p => p.Name == "Jane");

// We create a new object when we can't encounter a person
// above 60 years
// Person { Name: Some Name, Age: 62 }
var above60 = people.SingleOrDefault(
    p => p.Age > 60,
    new Person("Some Name", 62)
    );

record Person(string Name, int Age);
```

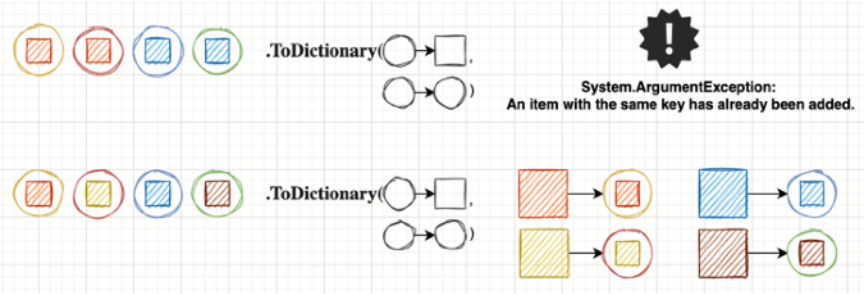
ToLookup



This methods creates a **lookup**. 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". This can be the object itself or a property of the object itself. At the end we have a list of distinct keys where the values share that exact key. 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")
};

// IGrouping<string, Product>
// [
//     "Electronic": [ "Smartphone", "PC"],
//     "Apple": [ "Fruit"]
// ]
var lookup = products.ToLookup(k => k.Category, elem => elem);

record Product(string Name, string Category);
```

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 (for example with the Add method).

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

// IGrouping<string, Product>
// [
//     1: Product { Id: 1, Name: "Smartphone" },
//     2: Product { Id: 2, Name: "PC" },
//     3: Product { Id: 3, Name: "Apple" }
// ]
var idToProductMapping = products.ToDictionary(k => k.Id, elem =>
elem);

// Product { Id: 1, Name: "Smartphone" }
var itemWithId1 = idToProductMapping[1];

record Product(int Id, string Name);
```

ToList / ToArray

As mentioned at the beginning, objects of the type Enumerable are not evaluated directly, but only when they are materialised. Beside quantifiers like Count or Sum there is also the possibility to pack the complete enumeration into a typed collection / array (**ToArray**) or list (**ToList**). With this we create the enumeration in memory at exactly this time.

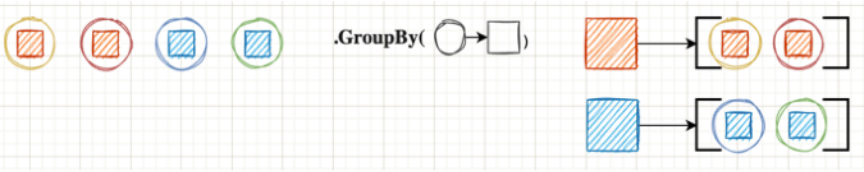
If we take the example from the beginning and call ToList directly, we see that the count does not change anymore.

```
var list = new List<int>();
list.Add(1);
list.Add(2);

var evenNumbers = list.Where(n => n % 2 == 0).ToList();

list.Add(4);
// This returns now only 1 as we materialised the list
Console.WriteLine($"Even numbers in list: evenNumbers.Count()}");
```

Grouping

This chapter will look into grouping capabilities of LINQ.

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. **GroupBy** means "I am building an object to represent the question 'what would these things look like if I organised them by group?'" 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")
};

// GroupBy creates an IEnumerable<IGrouping<string, Product>>
// This is a big difference to ToLookup where we don't have
// the "wrapping" IEnumerable
// [
//     "Electronic": [ "Smartphone", "PC"],
//     "Apple": [ "Fruit"]
// ]
var lookup = products.GroupBy(k => k.Category, elem => elem);

record Product(string Name, string Category);
```

Set

This chapter looks into functions, 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. Just imagine you have both lists together and call Distinct.

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

// [ 1, 2, 3, 4 ]
var result = numbers1.Union(numbers2);
```

Intersect

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

present in both will be in the result set. Also here: Only unique items are in the new list. Duplicates are automatically removed.

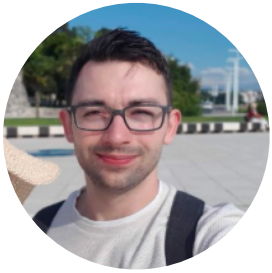
```
var numbers1 = new[] { 1, 1, 2 };
var numbers2 = new[] { 2, 3, 4 };

// [ 2 ]
var result = numbers1.Intersect(numbers2);
```

Real life samples

In this section you will find some “real life” examples which are more than just one method call. It consists out of runnable **dotnetfiddle** examples. Therefore you can just run the example or modify at your own will.

- Pagination of blog posts: https://dotnetfiddle.net/hsSIPV
- Best paid employee by department: https://dotnetfiddle.net/e2IfQu

Epilog
About Me



Hey I am Steven and author of that small "booklet". You can reach out via multiple channels I will list below. Any feedback welcome. Also newer version will come with more examples. So if you are missing something out, which I should add to the party, just let me know and I will update this little book.

Further Resources / Reads

- Generator-Function in C# - What does yield do?
- LINQMarbles - Interactive LINQ diagrams
- IEnumerable vs IQueryable - What's the difference
- Microsoft Documentation for Enumerable

Version

Version 1.3 (2023-12-03)

- Added LINQ-Marbles link

Version 1.22 (2023-03-27)

- Fixed that LINQ join behaves closer to SQL INNER join

Version 1.21 (2022-10-14)

- Naming is hard and therefore corrected a variable name

Version 1.2 (2022-09-16)

- Corrected code and explanation in introduction

Version 1.1 (2022-08-26)

- Corrected links

- Fixed Max throwing exception when empty
- Aggregate explanation for empty enumerable

Version 1.0 (2022-08-25)

- Initial Release