

LINQ

Explained with sketches

By Steven Giesel

# **Prólogo**

Esta pequena obra foi criada com o objetivo de dar aos iniciantes uma introdução visual e simples ao LINQ. Segundo o lema: "Imagens falam mais que mil palavras".

Qual é o objetivo deste pequeno livro? Ele deve permitir que você use os métodos do LINQ corretos na situação certa. Vou começar cada vez com um pequeno desenho, que será completado por uma pequena explicação, incluindo um exemplo de código. Vou abordar menos coisas como "LINQ-ToObject". Além disso, tocarei brevemente em coisas como **IQueryable** e **IEnumerable**.

Table of contents

Prologue 2

Table of contents 3

What is LINQ? 6

IEnumerable 7

Mindmap 9

Filtering 11

Where 11

Take 12

Skip 12

Distinct(By) 13

OfType 14

Projection 15

Select 15

SelectMany 15

Aggregation 17

Count 17

Aggregate 17

Max(By) 18

Quantification 19

Any 19

All 20

SequenceEquals 20

Merging 22

Join 22

Zip 23

Element 24

First 24

Single 24

FirstOrDefault / SingleOrDefault 25

Materialisation / Conversion 27

ToLookup 27

ToDictionary 28

ToList / ToArray 29

Grouping 30

GroupBy 30

Set 31

Union 31

Intersect 31

Real life samples 33

Epilog 34

About Me 34

Further Resources / Reads 35

Version 35

What is LINQ?

LINQ é a abreviação de “Language-Integrated Query” e é o nome de um conjunto de tecnologias baseadas na integração de recursos de consulta diretamente no C#. O objetivo é ter uma maneira uniforme e estruturada de operar em enumerações. As consultas LINQ retornam sempre o resultado como novos objetos. Isso garante que a enumeração original não será modificada. Isso é muito importante lembrar. Todas as consultas LINQ retornam uma nova enumeração em vez de excluir, atualizar ou adicionar novos itens ao determinado. Além disso, existem maneiras de transformar consultas LINQ em sintaxe SQL ou usar LINQ para passar por um documento XML. O tipo básico em que todas as consultas LINQ operam é IEnumerable.

IEnumerable

O tipo básico em que todas as consultas LINQ operam é o IEnumerable. Sem entrar em muitos detalhes, é fundamental entender que IEnumerable não representa uma lista “materializada”. Chamamos isso de “avaliação preguiçosa”. Isso significa que, no momento de executar as consultas LINQ, não obtemos os resultados reais. Somente quando enumeramos através da enumeração ou chamamos operações como ToList ou Count é que realmente “criamos” / “materializamos” o objeto.

Agora você verá um pequeno trecho. Não se preocupe se você não entender isso agora. Tome isso como uma motivação para entender completamente isso depois de ler o pequeno livro:

var list = new List<int>();

list.Add(1);

list.Add(2);

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

list.Add(4);

Console.WriteLine($"Números pares na lista: {evenNumbers.Count()}”);

Criamos a enumeração depois que a lista contém 2 elementos. Depois adicionamos outro número à própria lista. Então, quantos números pares temos na enumeração? A resposta é: 3. A razão é que nos materializamos em Count e não no momento de criar a enumeração em primeiro lugar. Então quando chamamos Count temos dois elementos, que são números pares (2 e 4). Nunca se esqueça disso.

Outro tipo sempre associado ao LINQ é IQueryable. IQueryable é basicamente IEnumerable com algo a mais e é exatamente essa parte do “algo a mais” que o torna tão único. Para isso, vou apenas listar os destaques aqui e fazer referência à minha postagem no blog:

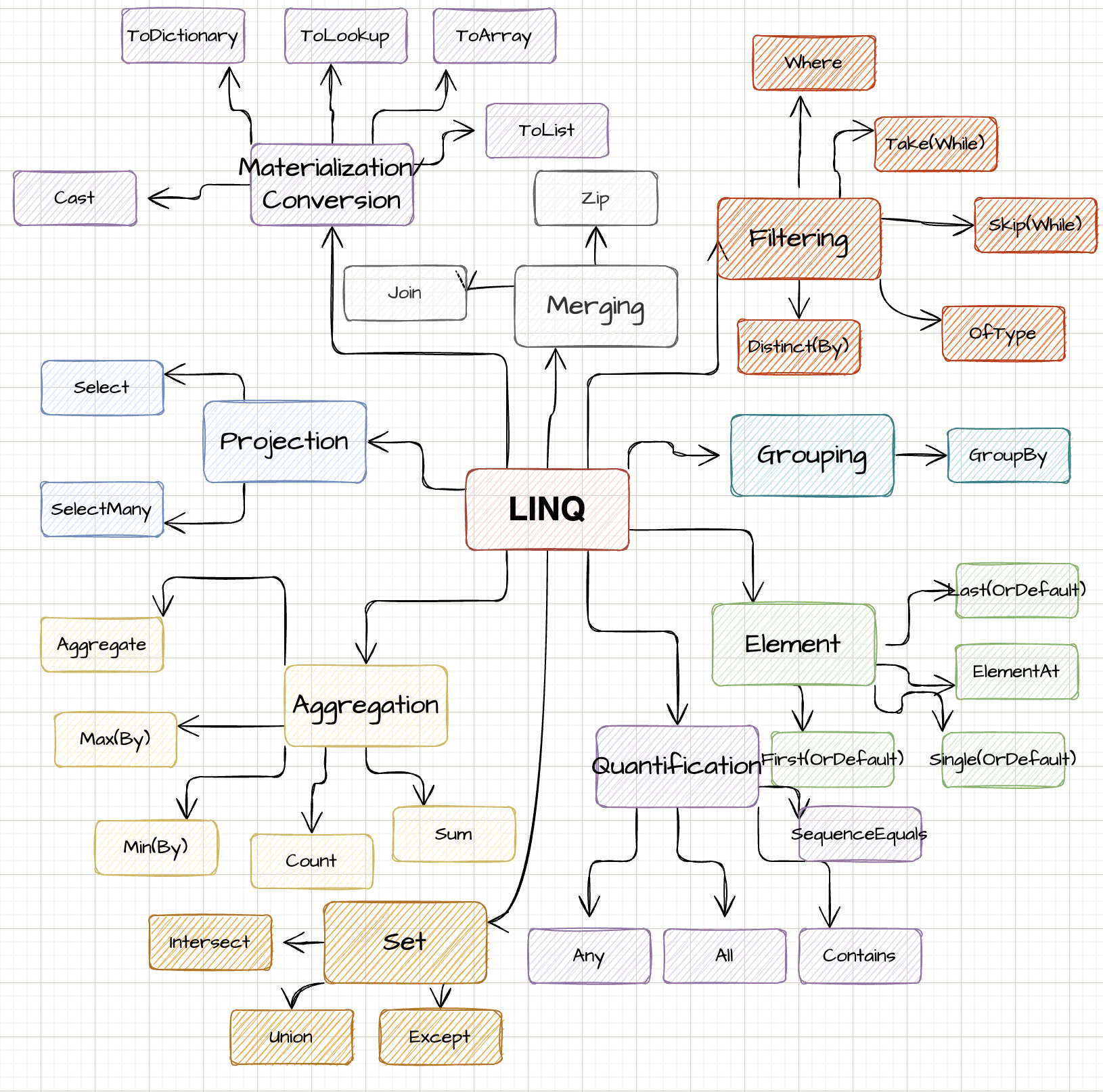
[IEnumerable vs IQueryable - What's the difference](https://steven-giesel.com/blogPost/606cb702-e391-4a26-9ae5-523eeb6196ed) no qual eu descrevo a diferença em detalhes.

* Tanto IEnumerable quanto IQueryable são coleções de encaminhamento - elas não são materializadas imediatamente
* A consulta de dados do banco de dados IEnumerable carregará os dados na memória no filtro posteriormente no cliente
* A consulta de dados do banco de dados IQueryable irá filtrar primeiro e depois enviar os dados filtrados para o cliente
* IQueryable é adequado para consultar dados de memória externa
* Pode haver cenários em que o provedor de consulta subjacente não pode traduzir sua expressão para algo significativo, então você deve mudar para IEnumerable

Mindmap

O LINQ tem muitas operações em seu conjunto de ferramentas para você. Assim, podemos agrupá-los em diferentes categorias. A próxima imagem mostrará uma visão geral aproximada para que você possa obter uma imagem mental. Eu também aconselharia voltar a essa imagem várias vezes para ver onde você está.

Os próximos capítulos são organizados exatamente por essas categorias.



O verdadeiro poder do LINQ surge quando você combina várias operações. Após a explicação dos operadores LINQ, você encontrará alguns exemplos do mundo real em que várias operações LINQ são usadas em uma instrução.

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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where?view=net-6.0)we can filter a given list based on our condition. The method accepts a [Predicate](https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-6.0). 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>();

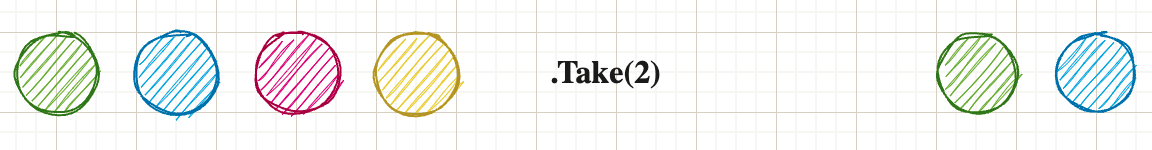
list.Add(1);

list.Add(2);

// Get even numbers

// Result: [ 2 ]

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

Take

[Take](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.take?view=net-6.0) 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 ]

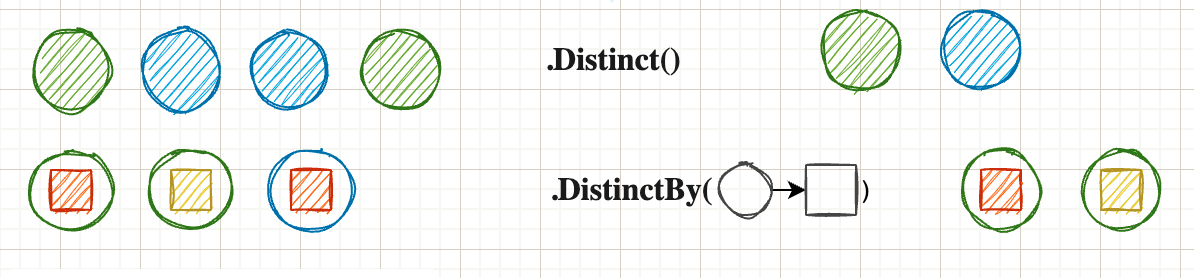
var takeOne = list.Take(1);

// Result: [ 1, 2 ]

var takeOneHundred = list.Take(100);

Skip

With [Skip](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skip?view=net-6.0) 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.

Distinct(By)

var list = new List<int>();

list.Add(1);

list.Add(2);

list.Add(3);

// Result: [ 2, 3 ]

var skipOne = list.Skip(1);

// Result: [ ]

var skipOneHundred = list.Take(100);

// Result: [ 2 ]

var pageTwo = list.Skip(1).Take(1);

[Distinct](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct?view=net-6.0) 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinctby?view=net-6.0) 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 list = new List<int>();

list.Add(1);

list.Add(1);

list.Add(2);

*// [ 1, 2 ]*

var uniqueElements = list.Distinct();

var people = new List<Person>

{

new Person("Steven", 31),

new Person("Katarina", 29),

new Person("Nils", 31)

};

*// [*

*// Person { Name = Steven, Age = 31 },*

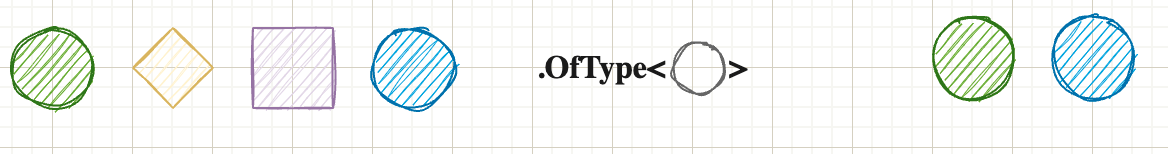
*// Person { Name = Katarina, Age = 29 }*

*// ]*

var uniqueAgedPeople = people.DistinctBy(p => p.Age);

record Person(string Name, int Age);

OfType



[OfType](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.oftype?view=net-6.0) 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>();

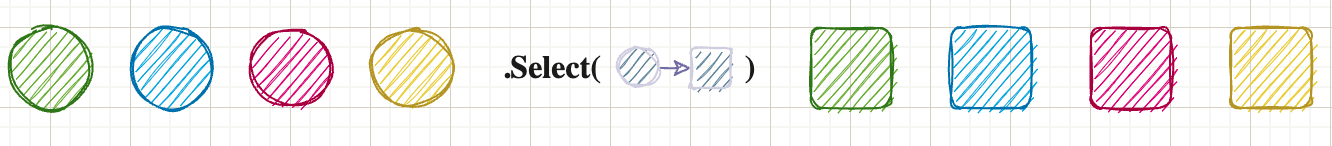
record Fruit;

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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=net-6.0) 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" },*

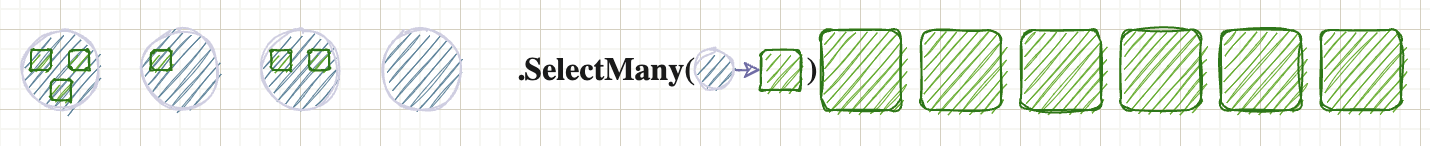
*// TargetObject { NumberAsString: "2" },*

*// ]*

var targetObjects = objects.Select(o => new TargetObject(o.ToString()));

record SourceObject(int Number);

record TargetObject(string NumberAsString);

SelectMany

[SelectMany](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-6.0) 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count?view=net-6.0) 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**](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.aggregate?view=net-6.0), 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.

Max(By)

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

*// 6*

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

*// 6*

var sumLinq = numbers.Sum();

[**Max**](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.max?view=net-6.0)[**(By)**](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.maxby?view=net-6.0) 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.any?view=net-6.0) 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),

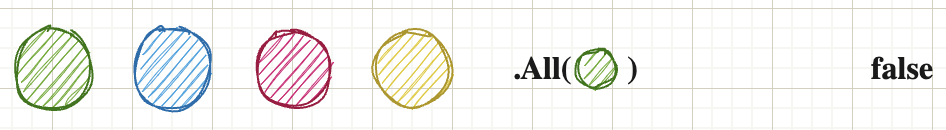
new Fruit("Apple", 51),

};

*// true*

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

record Fruit(string Name, int CaloriesPer100Gramm);

All

As the name implies checks if [All](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.all?view=net-6.0) 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),

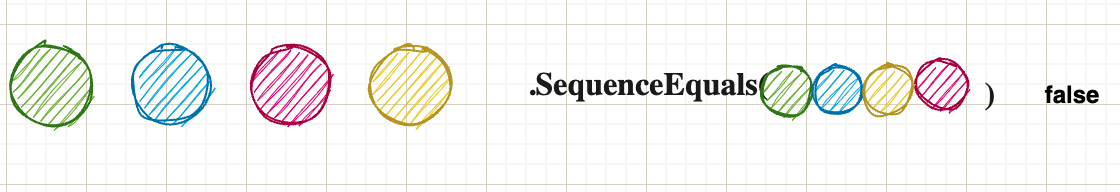
new Fruit("Apple", 51),

};

*// false*

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

record Fruit(string Name, int CaloriesPer100Gramm);

SequenceEquals

[SequenceEquals](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sequenceequal?view=net-6.0) 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 };

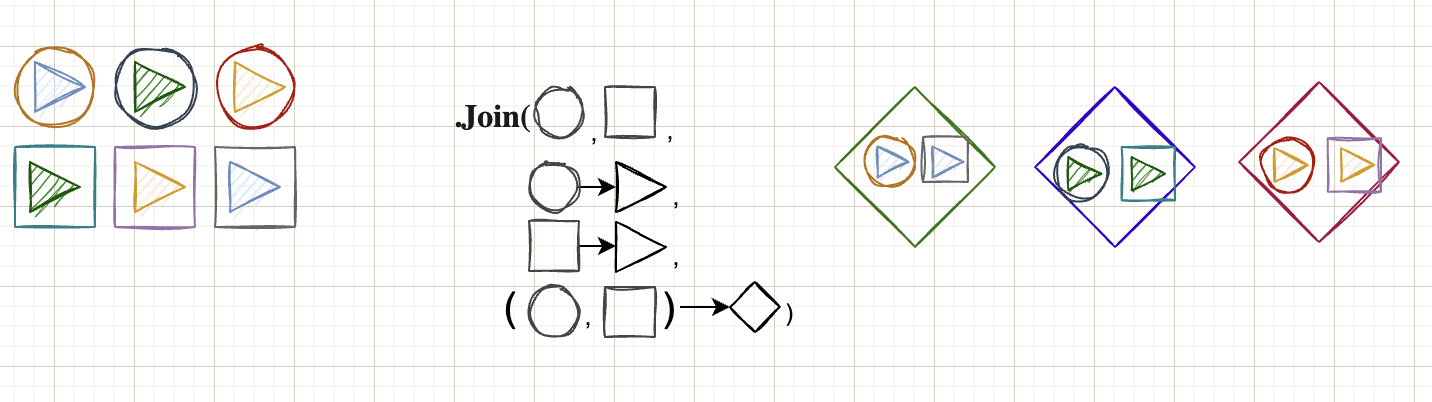
var moreNumbers = new[] { 1, 2, 4, 3 };

*// 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.join?view=net-6.0) works similar to a **SQL** Left-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.

Zip

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,

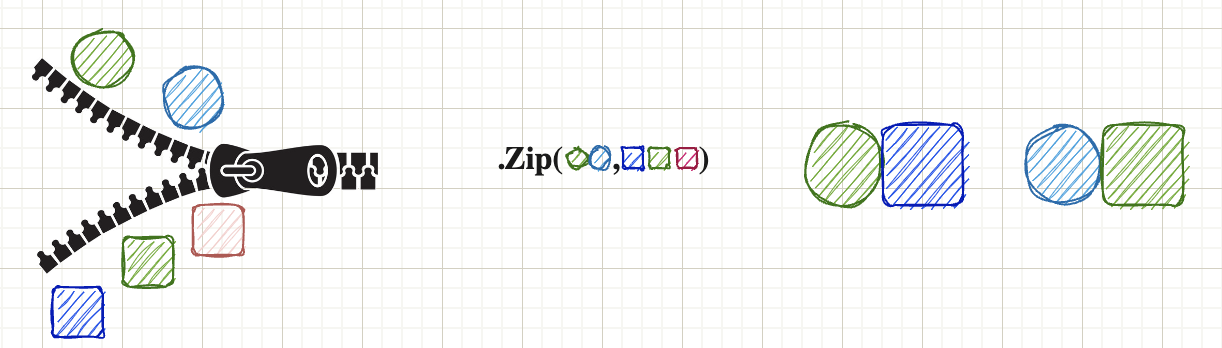
f => f.FruitId, c => c.FruitId,

(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);

With [Zip](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.zip?view=net-6.0) 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](http://www.apple.com/uk) 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),

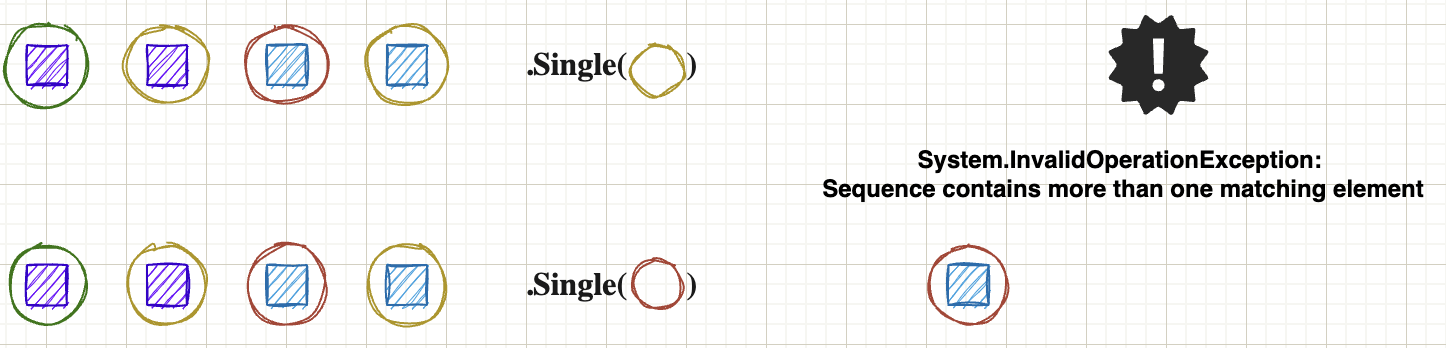
new Person("Dan", 28)

};

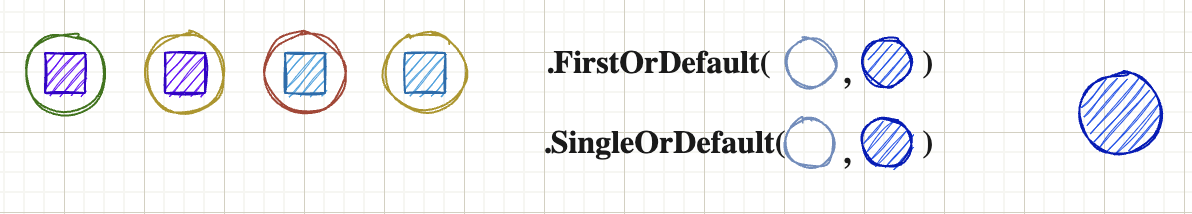
*// Person { Name: Steven, Age: 31 }*

var firstOver30 = people.First(p => p.Age > 30);

record Person(string Name, int Age);

Single

[Single](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.single?view=net-6.0) 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**.

FirstOrDefault /  
SingleOrDefault

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);

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(

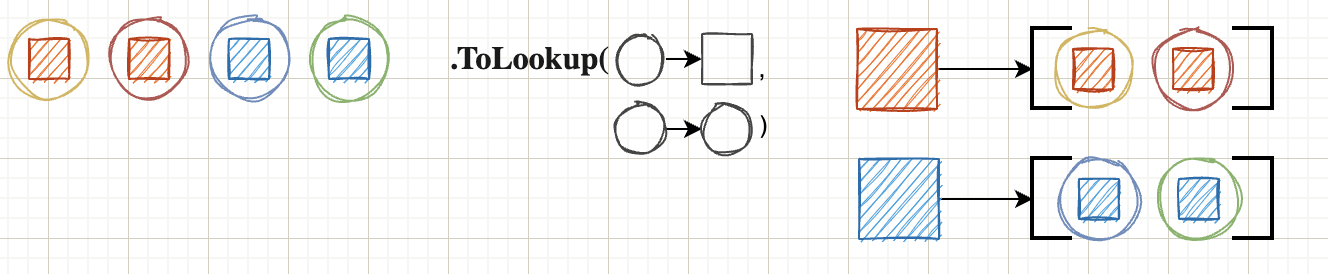
p => p.Age > 60,

new Person("Some Name", 62)

);

record Person(string Name, int Age);

Materialisation / Conversion



ToLookup

This methods creates a [lookup](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolookup?view=net-6.0). 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>*

*// [*

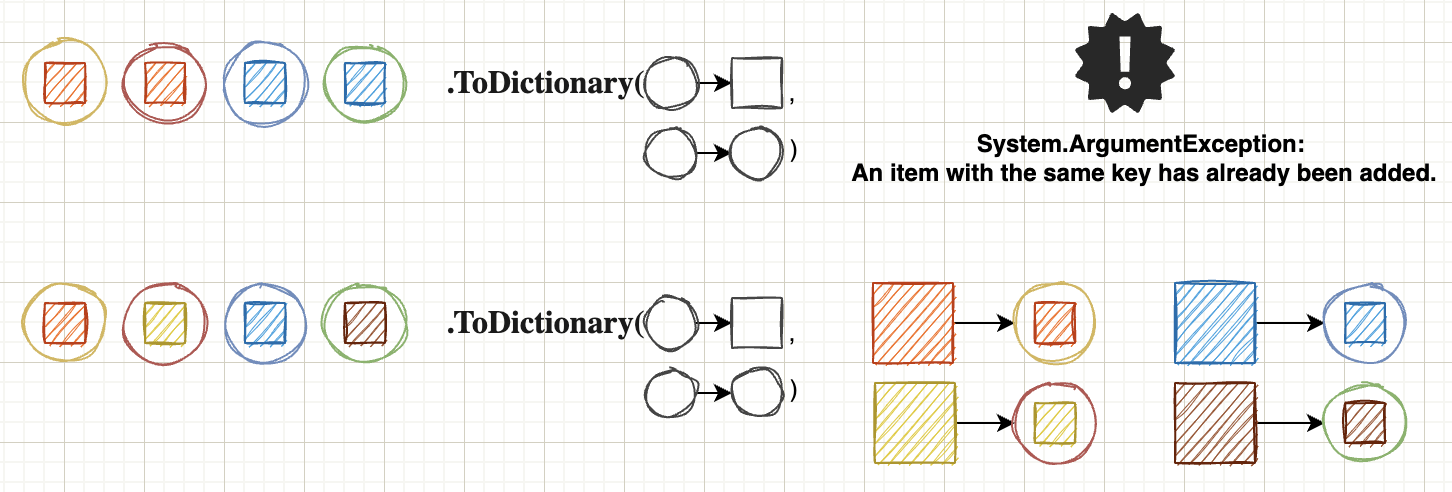
*// "Electronic": [ "Smartphone", "PC"],*

*// "Apple": [ "Fruit"]*

*// ]*

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

record Product(string Name, string Category);

ToDictionary

[ToDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.todictionary?view=net-6.0) 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.toarray?view=net-6.0)) or list ([ToList](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolist?view=net-6.0)). 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();

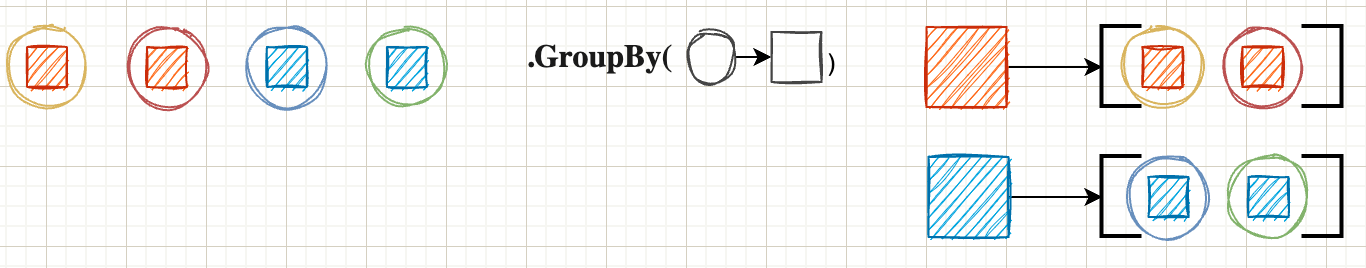
list.Add(4);

*// 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby?view=net-6.0) 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"]*

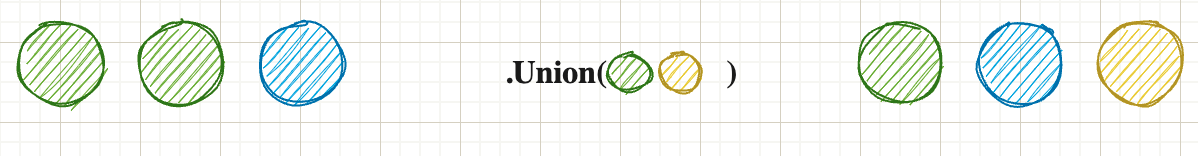
*// ]*

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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.union?view=net-6.0) 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](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.intersect?view=net-6.0) 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 Mepasted-image.tiff

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.[pasted-image.tiff](https://github.com/linkdotnet)[pasted-image.tiff](https://steven-giesel.com)[pasted-image.tiff](https://www.linkedin.com/in/steven-giesel/)

Further Resources / Reads

* [Generator-Function in C# - What does yield do?](https://steven-giesel.com/blogPost/994467f6-2429-4534-ad43-c0777076ab22)
* [IEnumerable vs IQueryable - What's the difference](https://steven-giesel.com/blogPost/994467f6-2429-4534-ad43-c0777076ab22)
* [Microsoft Documentation for Enumerable](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-6.0)

Version

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