.NET · LANGUAGE-INTEGRATED QUERY

LINQ

Explained with sketches

A visual, plain-language field guide to the LINQ query operators, every method paired with a diagram and a short C# example.

By Steven Giesel
BitSpire GmbH · Swiss Software Engineering

ABOUT US

Hey, we're BitSpire ;)

We are **.NET experts from Switzerland**, among us a Microsoft MVP and active open-source contributors. We have many years of experience building and maintaining systems and interfaces, mostly in cloud environments.

From advising CTOs on architecture through to hands-on .NET workshops, we love code, and a good deal more:

We <love> software that matters!

WHAT WE DO

- Custom Software
- Consulting
- Architecture
- Workshops

Want to work with us?

You can reach us anytime, no strings attached.

contact@bitspire.ch
bitspire.ch
linkedin.com/company/bitspire-ch

PROLOGUE

Pictures say more than a thousand words.

This little piece was created to give beginners a visual and simple introduction to LINQ. The goal is to enable you to reach for the *right* query in the right situation.

Each operator starts with a small drawing, completed by a short explanation and a code example. We go lightly on topics like `IQueryable` vs `IEnumerable`, just enough to be dangerous.

What is LINQ?

LINQ, short for *Language-Integrated Query*, is a set of technologies that bake query capabilities directly into C#. It gives you one uniform, structured way to operate on enumerations.

IMMUTABLE

Every query returns a *new* enumeration. The original is never mutated, nothing is added, updated or deleted in place.

ONE BASE TYPE

All LINQ queries operate on `IEnumerable`. Master it and the rest of the toolbox follows.

FOUNDATIONS

IEnumerable & lazy evaluation

`IEnumerable` does *not* represent a materialised list. This is **lazy evaluation**: at the moment you call a query you don't get results yet. Only when you enumerate, or call `ToList`, `Count`, etc., is the result actually created.

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

var evenNumbers = list.Where(n => n % 2 == 0);
list.Add(4);

// Prints 2, materialised on Count(), not on Where()
Console.WriteLine($"Even: {evenNumbers.Count()}");
```

The enumeration is built *when Count runs*, by then the list holds `1, 2, 4`, so two even numbers (2 and 4). Always keep the moment of materialisation in mind.

IEnumerable vs IQueryable

IEnumerable

Loads data into memory, then filters on the client. Forward-only, deferred. Your safe default for in-memory collections.

IQueryable

Filters *first* (e.g. translated to SQL), then sends only the result to the client. Ideal for out-of-memory sources like databases.

MINDMAP

The whole toolbox, grouped.

LINQ has many operators. Grouping them into categories gives you a mental map, the chapters that follow are organised in exactly this order. **Blue chips** are new in .NET 10.

FILTERING

Where · Take · Skip · Distinct(By) · OfType

PROJECTION

Select · SelectMany

AGGREGATION

Count · Aggregate · Max/Min(By) · Sum

QUANTIFICATION

Any · All · SequenceEqual

MERGING

Join · **LeftJoin** · **RightJoin** · Zip

ELEMENT

First · Single · …OrDefault

MATERIALISATION

ToLookup · ToDictionary · ToList/Array

GROUPING · SET

GroupBy · Union · Intersect

GENERATION

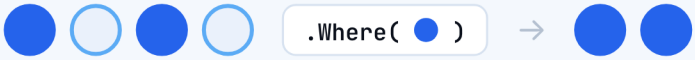
Sequence · **InfiniteSequence** - build sequences of numbers out of thin air, no source collection required.

FILTERING

Narrow an enumeration down based on a condition or a position.

Where

predicate → subset



Filters a list against a **predicate**, applied element by element. Every element for which it returns `true` ends up in the new enumeration.

```
var list = new List<int> { 1, 2 };
// Result: [ 2 ]
var evenNumbers = list.Where(n => n % 2 == 0);
```

Take

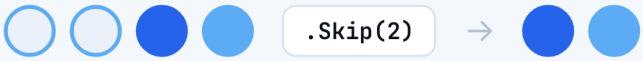
first N

Takes the given number of elements from the front. If fewer exist, you simply get what's left, no error. Pair it with **Skip** for pagination.

```
var list = new List<int> { 1, 2 };
var one  = list.Take(1);   // [ 1 ]
var many = list.Take(100); // [ 1, 2 ]
```

FILTERING 07 / 21

Skip drop first N



Skips the given number of elements. Skip more than the list holds and you get an empty enumeration back.

```
var list = new[] { 1, 2, 3, 4 };
// [ 3, 4 ]
var rest = list.Skip(2);
```

Distinct(By) remove duplicates

Returns a set with all duplicates removed. **DistinctBy** dedupes on a projected property instead of the whole object. Note: reference types compare by reference by default.

```
var nums = new[] { 1, 1, 2 };
var unique = nums.Distinct();          // [ 1, 2 ]// Steven(35) & Nils(35) share an age, one dropped
var byAge = people.DistinctBy(p => p.Age);
```

FILTERING → PROJECTION

OfType<T> keep by type

Keeps only elements that are of the given type (inherited types count too). Handy for untyped `object` arrays or pulling one subclass out of a mixed list.

```
var fruits = new List<Fruit> { new Banana(), new Apple() };
// [ Apple { } ]
var apples = fruits.OfType<Apple>();
```

PROJECTION — transform items into a new shape

Select map 1 → 1

Projects each item into a new form, mapping one type to another. The result set always has the **same count** as the source.

```
var objects = new[] { new Source(1), new Source(2) };
// [ "1", "2" ]
var mapped = objects.Select(o => o.Number.ToString());
```

PROJECTION → AGGREGATION

SelectMany `flatten`

Flattens a list-of-lists into a single, one-dimensional sequence.

```
var recipes = new[] {
    new Recipe("Pizza",     ["Tomato", "Basil"]),
    new Recipe("Hot Water", ["Water"]),
};
// [ "Tomato", "Basil", "Water" ]
var allIngredients = recipes.SelectMany(r => r.Ingredients);
```

AGGREGATION — reduce a whole list to one value

Count `→ number`

Counts the elements, optionally only those matching a predicate.

```
var names = new[] { "Steven", "Marie", "Steven" };
// 2
var stevens = names.Count(n => n == "Steven");
```

AGGREGATION

Aggregate

reduce / fold

Also known as *reduce*: folds every element into a single scalar, starting from a seed value. An empty sequence returns the seed unchanged.

```
var numbers = new[] { 1, 2, 3 };
var sum = numbers.Aggregate(0, (curr, next) => curr + next); // 6
var same = numbers.Sum();                                    // 6
```

Max(By) · Min(By)

extreme element

Retrieves the largest (or with **MinBy**, the smallest) element by a projected key. On an empty sequence it throws.

```
var people = new[] { new Person("Steven", 35), new Person("Jean", 22) };
// Person { Name: Steven, Age: 35 }
var oldest = people.MaxBy(p => p.Age);
```

QUANTIFICATION

Measure the quantity of matching elements, these return a bool.

Any at least one?

Returns `true` the moment one element satisfies the condition, then stops. Empty sequence → `false`.

```
var fruits = new[] { new Fruit("Banana", 89), new Fruit("Apple", 51) };
// true
var hasDense = fruits.Any(f => f.CaloriesPer100g > 80);
```

All every one?

Returns `true` only if *every* element matches. On the first miss it stops and returns `false`.

```
// false, the apple has only 51 kcal
var allDense = fruits.All(f => f.CaloriesPer100g > 80);
```

QUANTIFICATION → MERGING

SequenceEqual

same order & values?

Checks that two sequences have the same length *and* equal elements in the same order (default comparer; an `IEqualityComparer` is optional). Two empty lists are equal.

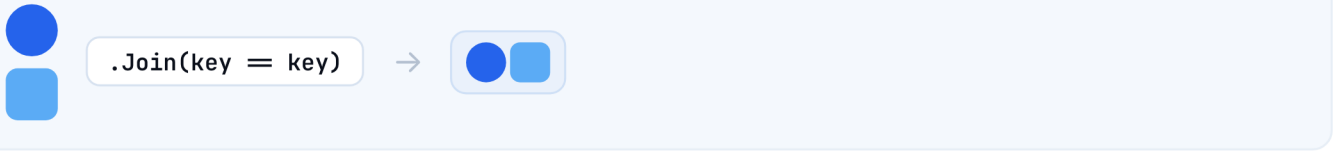
```
var a = new[] { 1, 2, 3, 4 };
var b = new[] { 1, 2, 4, 3 };
// false, same values, different order
var equal = a.SequenceEqual(b);
```

MERGING

combine two sequences into one

Join

SQL inner join



Like a SQL inner join: matches each element of A against B on a key. Where keys are equal, a result element is built from the pair. **Unmatched elements are dropped.**

```
// { Name = Banana, Class = Magnesium-rich }
var joined = fruits.Join(classes,
    f => f.FruitId, c => c.FruitId,
    (f, c) => new { f.Name, c.Class });
```

MERGING · OUTER JOINS

Where `Join` drops the unmatched, outer joins keep one whole side and fill the other with `null`.

LeftJoin

keep all left

Every element of the **left** (outer) sequence is returned, matched or not. When no match exists, the inner element is `null`, so guard it in your result selector.

```
Person[] people = [new("Ana"), new("Bob"), new("Cleo")];
Pet[]    pets   = [new("Ana","Rex"), new("Cleo","Miez")];

// Ana → Rex · Bob → (no pet) · Cleo → Miez
var owners = people.LeftJoin(pets,
    p => p.Name, pet => pet.Owner,
    (p, pet) => $"{p.Name} → {pet?.Name ?? "(no pet)"}");
```

RightJoin

keep all right

The mirror image: every element of the **right** (inner) sequence is kept, and the *outer* element is `null` when nothing matches. Fido's owner isn't in the list, so his person is null.

```
Person[] people = [new("Ana"), new("Cleo")];
Pet[]    pets   = [new("Ana","Rex"), new("Bob","Fido")];

// Ana → Rex · (no owner) → Fido
var owners = people.RightJoin(pets,
    p => p.Name, pet => pet.Owner,
    (p, pet) => $"{p?.Name ?? "(no owner)"} → {pet.Name}");
```

MERGING → ELEMENT

Zip

pair by position

Merges two lists element-by-element with a combining function, stopping as soon as *either* lane runs out, so the result is as long as the shorter input.

```
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

pull a single item out

First

first match

Returns the first element (optionally matching a predicate) and stops immediately. If nothing is found, it **throws**.

```
// Person { Name: Steven, Age: 35 }
var firstOver30 = people.First(p => p.Age > 30);
```

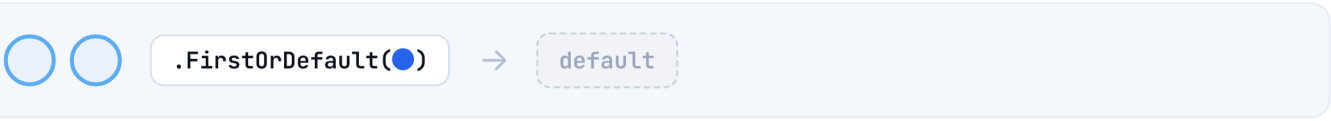
ELEMENT

Single exactly one

Unlike **First**, **Single** scans the whole sequence to guarantee there is exactly one match. It throws if it finds a second, or if it finds none.

```
// Person { Name: Steven, Age: 35 }
var steven = people.Single(p => p.Name == "Steven");

// throws, more than one person is over 30
var over30 = people.Single(p => p.Age > 30);
```

FirstOrDefault · SingleOrDefault



When nothing is found these return the *default* instead of throwing, `null` for reference types, `0` for numbers. Since .NET 6 you can supply your own default.

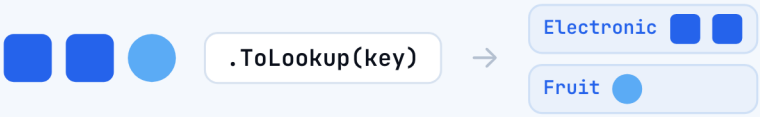
```
// null, nobody named Jane
var jane = people.FirstOrDefault(p => p.Name == "Jane");

// custom fallback when nobody is over 60
var senior = people.SingleOrDefault(
    p => p.Age > 60, new Person("N/A", 62));
```

MATERIALISATION · CONVERSION

Turn a lazy query into a concrete, in-memory structure.

ToLookup

key → many (1:n)



Builds an immutable lookup where each key maps to a *list* of values. Materialised eagerly, the whole thing is computed right now.

```
// Electronic → [Smartphone, PC] · Fruit → [Apple]
var lookup = products.ToLookup(p => p.Category);
```

ToDictionary

key → one (1:1)

Like ToLookup but strictly one-to-one and **mutable**. Two items sharing a key throws `ArgumentException`.

```
var products = new[] { new Product(1,"Smartphone"), new Product(2,"PC") };
var byId = products.ToDictionary(p => p.Id);
var pc = byId[2];   // Product { Id: 2, Name: PC }
```

CONVERSION → GROUPING

ToList · ToArray

materialise now

Packs the enumeration into a concrete collection *at this exact moment*. Later changes to the source no longer affect it, the query is frozen.

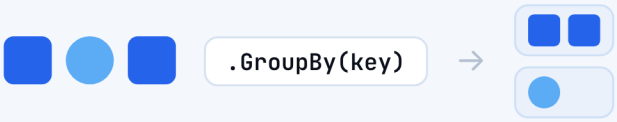
```
var list = new List<int> { 1, 2 };
var evens = list.Where(n => n % 2 == 0).ToList();
list.Add(4);
// 1, materialised before 4 was added
Console.WriteLine(evens.Count);
```

GROUPING

organise items by a shared key

GroupBy

deferred groups



Groups items by a key into an `IEnumerable<IGrouping>`. Almost identical to ToLookup, but **deferred**, it describes the grouping rather than caching it right away.

```
// IEnumerable<IGrouping<string, Product>>
var groups = products.GroupBy(p => p.Category);
```

SET

Set operations always return distinct elements, duplicates are removed automatically.

Union A ∪ B

Every distinct element present in *either* list. Think: concatenate both, then **Distinct**.

```
var a = new[] { 1, 1, 2 };
var b = new[] { 2, 3, 4 };
// [ 1, 2, 3, 4 ]
var result = a.Union(b);
```

Intersect A ∩ B

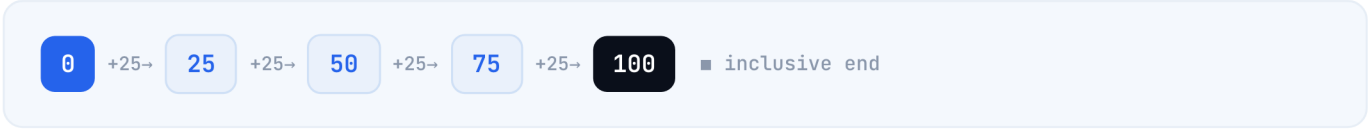
Only the distinct elements present in *both* lists.

```
// [ 2 ]
var common = a.Intersect(b);
```

GENERATION

Most operators consume a source collection. These two *produce* one, a sequence of numbers built from a start, a step and (maybe) an end. They live on `Enumerable`, not on an instance.

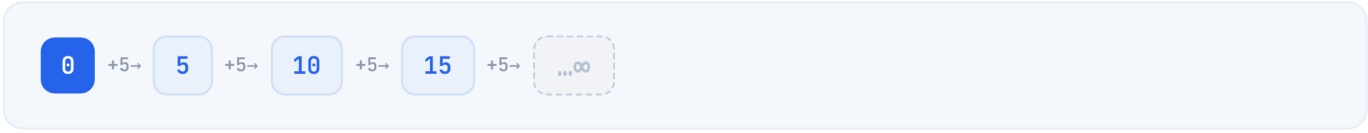
Sequence `start → endInclusive, step`



Generates numbers from `start` up to and including `endInclusive`, incrementing by `step`. Works with any numeric type (`INumber<T>`), `int`, `double`, `decimal`…

```
// 0, 25, 50, 75, 100
var scale = Enumerable.Sequence(0, 100, 25);

// 1, 3, 5, 7, 9 , odd numbers up to 10
var odds = Enumerable.Sequence(1, 10, 2);
```

InfiniteSequence `start, step, forever`



Never stops, it yields values lazily and endlessly. On its own it's a definition, not a computation.

⚠ ALWAYS BOUND IT
Pair with `Take`, `First` or `TakeWhile`, enumerate it fully and it loops forever.

```
// endless: 0, 5, 10, 15, 20, …
var stream = Enumerable.InfiniteSequence(0, 5);

// bound it, 0, 5, 10, 15
var first4 = Enumerable.InfiniteSequence(0, 5).Take(4);
```

REAL-LIFE SAMPLES

The real power is composition.

A single operator is handy; chaining them is where LINQ shines. Here is paginated, published blog posts sorted newest-first, five operators in one readable statement.

```
IEnumerable<BlogPost> posts = await GetAllBlogPosts();

var page2 = posts
    .Where(p => p.IsPublished)
    .OrderByDescending(p => p.PublishDate)
    .Skip(pageSize * (page - 1))
    .Take(pageSize)
    .ToList();
```

▷ Pagination of blog posts — dotnetfiddle.net/hsSIPV

▷ Best-paid employee by dept — dotnetfiddle.net/e2IfQu

Further resources

Microsoft docs, System.Linq.Enumerable — learn.microsoft.com ↗

IEnumerable vs IQueryable · yield explained — steven-giesel.com ↗

IN CLOSING

Want to go further?

LINQ is only one corner of modern .NET. There is plenty more worth getting right: async streams, high-performance spans, source generators, resilient HTTP and clean architecture.

Every codebase is different, and that is exactly why a short conversation about the right approach beats an answer off the shelf.

Discuss your project with us?

Drop us a line, we would love to hear from you.

contact@bitspire.ch
bitspire.ch