LINQ Methods

In LINQ

Sai Reddy 4/2/25 .NET

LINQ Methods

LINQ (**Language Integrated Query**) provides a **powerful and readable** way to work with collections and databases in C#. Let's break down each **LINQ method** into categories with examples.

1. Filtering Methods

Filtering methods are used to extract elements that meet specific conditions.

Where() – Filter by Condition

- **Usage**: Filters elements based on a condition.
- **Parameters**: A predicate function (Func<T, bool>) to apply on each element.

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };
var evenNumbers = numbers.Where(n => n % 2 == 0);
Console.WriteLine(string.Join(", ", evenNumbers));
```

Output: 2, 4

OfType() – Filter by Type

- **Usage**: Filters elements based on their type.

```
var mixedList = new List<object> { 1, "Hello", 2, "World" };
var strings = mixedList.OfType<string>();
Console.WriteLine(string.Join(", ", strings));
```

Output: Hello, World

2. Projection Methods

Projection methods transform data from one form to another.

Select() – Transform Elements

- **Usage**: Maps each element to a new form.

```
var numbers = new List<int> { 1, 2, 3 };
var squares = numbers.Select(n => n * n);
Console.WriteLine(string.Join(", ", squares));
```

Output: 1, 4, 9

SelectMany() – Flatten Nested Collections

- **Usage**: Projects each element into multiple elements.

```
var words = new List<string> { "Hello", "World" };
var letters = words.SelectMany(w => w);
Console.WriteLine(string.Join(", ", letters));
```

Output: H, e, l, l, o, W, o, r, l, d

3. Ordering Methods

Sorting methods arrange elements in a sequence.

OrderBy() – Ascending Order

```
var numbers = new List<int> { 3, 1, 4 };
var sorted = numbers.OrderBy(n => n);
Console.WriteLine(string.Join(", ", sorted));
```

Output: 1, 3, 4

OrderByDescending() – Descending Order

```
var sortedDesc = numbers.OrderByDescending(n => n);
Console.WriteLine(string.Join(", ", sortedDesc));
```

Output: 4, 3, 1

ThenBy() – Secondary Sorting

```
var people = new List<(string Name, int Age)>
{
   ("Alice", 25), ("Bob", 20), ("Alice", 20)
};

var sorted = people.OrderBy(p => p.Name).ThenBy(p => p.Age);
foreach (var person in sorted) Console.WriteLine($"{person.Name} {person.Age}");
```

Output:

```
Alice 20
Alice 25
Bob 20
```

ThenByDescending() – Secondary Descending Sorting

```
var sortedDesc = people.OrderBy(p => p.Name).ThenByDescending(p => p.Age);
```

Output:

```
Alice 25
Alice 20
Bob 20
```

Reverse() – Reverse Order

```
var reversed = numbers.AsEnumerable().Reverse();
Console.WriteLine(string.Join(", ", reversed));
```

Output: 4, 1, 3

4. Grouping Methods

Grouping methods organize elements based on a key.

GroupBy() – Group Elements

```
var people = new List<string> { "Alice", "Bob", "Anna" };
var grouped = people.GroupBy(p => p[0]);

foreach (var group in grouped)
{
   Console.WriteLine($"{group.Key}: {string.Join(", ", group)}");
}
```

Output:

```
A: Alice, Anna
B: Bob
```

ToLookup() – Creates a Lookup Table

Similar to GroupBy(), but returns a Lookup<TKey, TElement>.

```
var lookup = people.ToLookup(p => p[0]);
Console.WriteLine(string.Join(", ", lookup['A']));
```

Output: Alice, Anna

5. Set Operations

Used for **unique elements** and **comparisons**.

Distinct() – Remove Duplicates

```
var numbers = new List<int> { 1, 2, 2, 3 };
var uniqueNumbers = numbers.Distinct();
Console.WriteLine(string.Join(", ", uniqueNumbers));
```

Output: 1, 2, 3

Except() – Elements in First but Not Second

```
var set1 = new List<int> { 1, 2, 3 };
var set2 = new List<int> { 2, 3, 4 };
var result = set1.Except(set2);
Console.WriteLine(string.Join(", ", result));
```

Output: 1

Intersect() – Common Elements

```
var result = set1.Intersect(set2);
```

Output: 2, 3

Union() – Unique Elements from Both

```
var result = set1.Union(set2);
```

Output: 1, 2, 3, 4

Concat() – Merge Lists

```
var result = set1.Concat(set2);
```

Output: 1, 2, 3, 2, 3, 4

6. Element Access Methods

Methods to retrieve elements.

First()

- **Usage**: Returns the first element of a sequence.
- **Parameters**:
  - Func<T, bool> (optional) → A condition to match.
- **Return Type**: T (element type)
- **Throws Exception**: If no elements are found.

```
var numbers = new List<int> { 10, 20, 30 };
Console.WriteLine(numbers.First()); // Output: 10
```

FirstOrDefault()

- **Usage**: Like First(), but returns a **default value** if no element is found.
- **Return Type**: T or default(T)

```
var emptyList = new List<int>();
Console.WriteLine(emptyList.FirstOrDefault()); // Output: 0 (default int value)
```

Last()

- **Usage**: Returns the last element.
- **Throws Exception**: If the sequence is empty.

```
Console.WriteLine(numbers.Last()); // Output: 30
```

LastOrDefault()

- **Usage**: Like Last(), but returns **default value** if empty.

```
Console.WriteLine(emptyList.LastOrDefault()); // Output: 0
```

Single()

- **Usage**: Returns a **single element** from a collection.
- **Throws Exception**: If there are **none or more than one** element.

```
var singleElementList = new List<int> { 42 };
Console.WriteLine(singleElementList.Single()); // Output: 42
```

SingleOrDefault()

- **Usage**: Like Single(), but returns **default value** if no elements exist.

```
Console.WriteLine(emptyList.SingleOrDefault()); // Output: 0
```

ElementAt()

- **Usage**: Retrieves an element at a **specific index**.

- **Throws Exception**: If index is **out of range**.

```
Console.WriteLine(numbers.ElementAt(1)); // Output: 20
```

ElementAtOrDefault()

- **Usage**: Like ElementAt(), but returns **default value** if index is **out of range**.

```
Console.WriteLine(numbers.ElementAtOrDefault(10)); // Output: 0
```

DefaultIfEmpty()

- **Usage**: Returns a **default value** if the sequence is **empty**.

```
var result = emptyList.DefaultIfEmpty(100);
Console.WriteLine(string.Join(", ", result)); // Output: 100
```

7. Aggregation Methods

Methods to calculate results.

Sum(), Count(), Min(), Max(), Average()

```
Console.WriteLine(numbers.Sum()); // 6
Console.WriteLine(numbers.Count()); // 3
Console.WriteLine(numbers.Min()); // 1
Console.WriteLine(numbers.Max()); // 3
Console.WriteLine(numbers.Average()); // 2
```

Aggregate()

- **Usage**: Applies an **accumulator function**.

```
var sum = numbers.Aggregate((total, next) => total + next);
Console.WriteLine(sum); // Output: 60
```

8. Joining Methods

Methods to combine data.

Join()

```
var result = people.Join(orders, p => p.Id, o => o.PersonId, (p, o) => new { p.Name, o.Product });
```

GroupJoin()

- **Usage**: Groups and joins two sequences based on a **key**.
- **Parameters**:
  - outer → First sequence.
  - inner → Second sequence.
  - outerKeySelector → Key from outer.
  - innerKeySelector → Key from inner.
  - resultSelector → Combines matched elements.

```
var persons = new List<Person>
{
   new Person { Id = 1, Name = "Alice" },
   new Person { Id = 2, Name = "Bob" }
};

var orders = new List<Order>
{
   new Order { PersonId = 1, Product = "Laptop" },
   new Order { PersonId = 2, Product = "Phone" },
   new Order { PersonId = 1, Product = "Tablet" }
};

var result = persons.GroupJoin(
   orders,
   p => p.Id,
   o => o.PersonId,
   (p, orders) => new { p.Name, Orders = orders.Select(o => o.Product) }
);

foreach (var entry in result)
{
   Console.WriteLine($"{entry.Name}: {string.Join(", ", entry.Orders)}");
}
```

Output:

```
Alice: Laptop, Tablet
Bob: Phone
```

9. Quantification Methods

These methods check conditions on collections.

Any()

- **Usage**: Checks if **any element** exists or satisfies a condition.
- **Return Type**: bool

```
Console.WriteLine(numbers.Any()); // Output: True
Console.WriteLine(numbers.Any(n => n > 25)); // Output: True
```

All()

- **Usage**: Checks if **all elements** satisfy a condition.

```
Console.WriteLine(numbers.All(n => n > 5)); // Output: True
Console.WriteLine(numbers.All(n => n > 20)); // Output: False
```

10. Partitioning Methods

These methods extract parts of a sequence.

Skip()

- **Usage**: Skips the **first N elements**.

```
var skipped = numbers.Skip(2);
Console.WriteLine(string.Join(", ", skipped)); // Output: 30
```

Take()

- **Usage**: Takes the **first N elements**.

```
var taken = numbers.Take(2);
Console.WriteLine(string.Join(", ", taken)); // Output: 10, 20
```

SkipWhile()

- **Usage**: Skips elements **while** a condition is **true**.

```
var skippedWhile = numbers.SkipWhile(n => n < 20);
Console.WriteLine(string.Join(", ", skippedWhile)); // Output: 20, 30
```

TakeWhile()

- **Usage**: Takes elements **while** a condition is **true**.

```
var takenWhile = numbers.TakeWhile(n => n < 30);
Console.WriteLine(string.Join(", ", takenWhile)); // Output: 10, 20
```

11. Conversion Methods

Convert sequences into **different types**.

ToArray()

- **Usage**: Converts a collection to an **array**.

```
var array = numbers.ToArray();
Console.WriteLine(array.Length); // Output: 3
```

ToList()

- **Usage**: Converts a collection to a **List**.

```
var list = numbers.ToList();
Console.WriteLine(list.Count); // Output: 3
```

ToDictionary()

- **Usage**: Converts a collection to a **Dictionary<TKey, TValue>**.
- **Parameters**:
  - Func<T, TKey> → Key selector.
  - Func<T, TValue> (optional) → Value selector.

```
var dict = numbers.ToDictionary(n => n, n => $"Number {n}");
Console.WriteLine(dict[10]); // Output: Number 10
```

Cast()

- **Usage**: Converts a **non-generic** collection into a **specific type**.

```
var objList = new ArrayList { 1, 2, 3 };
var intList = objList.Cast<int>().ToList();
Console.WriteLine(string.Join(", ", intList)); // Output: 1, 2, 3
```