8. Language Integrated Query (LINQ)

Contents

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a **first-class language construct**, just like classes, methods, events. You **write queries against strongly typed collections of objects** by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML) [1].

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform **filtering**, **ordering**, and **grouping** operations on data sources with a minimum of code.

You can write LINQ queries in C# for **any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface**.

A *query* is an expression that retrieves data from a data source. LINQ offers a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, .NET collections and any other format for which a LINQ provider is available. [2]

8.1 Three Parts of a Query Operation

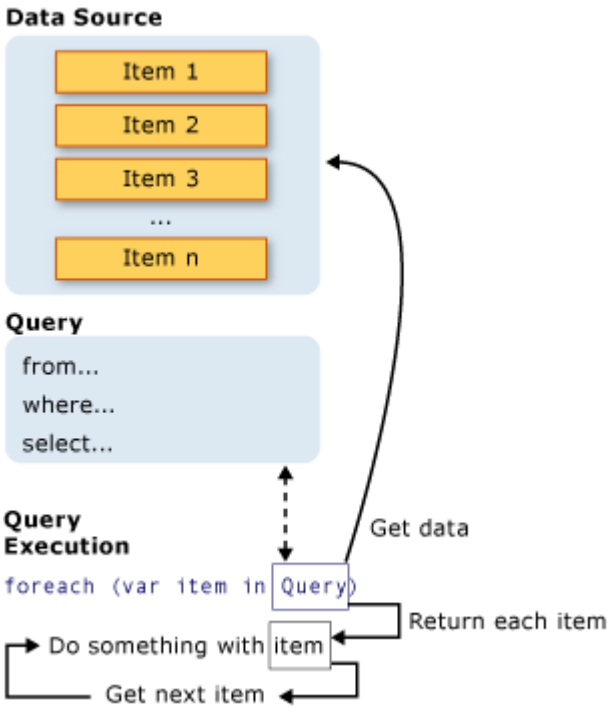
All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also. This example is referred to throughout the rest of this topic.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

The following illustration shows the complete query operation. In LINQ, the execution of the query is distinct from the query itself. In other words, you have not retrieved any data just by creating a query variable.



8.1.1 The Data Source

In the previous example, because the data source is an array, it implicitly supports the generic IEnumerable<T> interface. This fact means it can be queried with LINQ. A query is executed in a `foreach` statement, and `foreach` requires IEnumerable or IEnumerable<T>. Types that support IEnumerable<T> or a derived interface such as the generic IQueryable<T> are called *queryable types*.

8.1.2 The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a **query expression**. To make it easier to write queries, C# has introduced new **query syntax**.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where` and `select`. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. It is important to know that in LINQ, t**he query variable itself takes no action and returns no data**. It just stores the information that is required to produce the results when the **query is executed at some later point**.

Note: queries can also be expressed by using method syntax. More about this can be read further in the document

8.1.3 Query Execution

8.1.3.1 Deferred Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement. This concept is referred to as *deferred execution* and is demonstrated in the following example:

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

The `foreach` statement is where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

8.1.3.2 Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. Note also that these types of queries return a single value, not an `IEnumerable` collection. The following query returns a count of the even numbers in the source array:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the ToList or ToArray methods.

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

You can also force execution by putting the `foreach` loop immediately after the query expression. However, by calling `ToList` or `ToArray` you also cache all the data in a single collection object.

8.2 Query expression syntax

A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL. Each clause in turn contains one or more C# expressions, and these expressions may themselves be either a query expression or contain a query expression.

A query expression must begin with a from clause and must end with a select or group clause. Between the first `from` clause and the last `select` or `group` clause, it can contain one or more of these optional clauses: where, orderby, join, let and even additional from clauses. You can also use the into keyword to enable the result of a `join` or `group` clause to serve as the source for additional query clauses in the same query expression.

8.2.1 from clause

A query expression must begin with a `from` clause. The `from` clause specifies the following:

- The data source on which the query will be run.
- A local *range variable* that represents each element in the source sequence.

Both the range variable and the data source are strongly typed. The data source referenced in the `from` clause must have a type of IEnumerable, IEnumerable<T>, or a derived type such as IQueryable<T>.

In the following example, `numbers` is the data source and `num` is the range variable. Note that both variables are strongly typed even though the var keyword is used.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

8.2.1.1 The range variable

The compiler infers the type of the range variable when the data source implements IEnumerable<T>. For example, if the source has a type of `IEnumerable<Customer>`, then the range variable is inferred to be `Customer`.

In the previous example `num` is inferred to be of type `int`. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing `select num`, you could write `select num.ToString()` to cause the query expression to return a sequence of strings instead of integers. Or you could write `select num + 10` to cause the expression to return the sequence 14, 11, 13, 12, 10.

The range variable is like an iteration variable in a foreach statement except for one very important difference: a range variable never actually stores data from the source. It's just a syntactic convenience that enables the query to describe what will occur when the query is executed.

8.2.2 where clause

The `where` clause is used in a query expression to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (*predicate*) to each source element (referenced by the range variable) and returns those for which the specified condition is true. A single query expression may contain multiple `where` clauses and a single clause may contain multiple predicate subexpressions.

8.2.2.1 Example 1

In the following example, the `where` clause filters out all numbers except those that are less than five. If you remove the `where` clause, all numbers from the data source would be returned. The expression `num < 5` is the predicate that is applied to each element.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

8.2.2.2 Example 2

Within a single `where` clause, you can specify as many predicates as necessary by using the && and || operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

8.2.2.3 Example 3

A `where` clause may contain one or more methods that return Boolean values. In the following example, the `where` clause uses a method to determine whether the current value of the range variable is even or odd.

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

8.2.2.4 Remarks

The `where` clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A `where` clause may appear either before or after a group clause depending on whether you have to filter the source elements before or after they are grouped.

If a specified predicate is not valid for the elements in the data source, a compile-time error will result. This is one benefit of the strong type-checking provided by LINQ.

8.2.3 select clause

In a query expression, the `select` clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the `select` clause itself. A query expression must terminate with either a `select` clause or a group clause.

The following example shows a simple `select` clause in a query expression.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

The type of the sequence produced by the `select` clause determines the type of the query variable `queryHighScores`. In the simplest case, the `select` clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. However, the `select` clause also provides a powerful mechanism for transforming (or *projecting*) source data into new types.

8.2.3.1 Selecting a subset of each source element

There are two primary ways to select a subset of each element in the source sequence:

1. To select just **one member** of the source element, use the dot operation. In the following example, assume that a `Customer` object contains several public properties including a string named `City`. When executed, this query will produce an output sequence of strings.

```
var query = from cust in Customers
            select cust.City;
```

2. To create elements that contain more than one property from the source element, you can use an **object initializer** with either a **named object or an anonymous type**. The following example shows the use of an anonymous type to encapsulate two properties from each `Customer` element:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

8.2.4 join clause

The `join` clause is useful for associating elements from different source sequences that have no direct relationship in the object model. The only requirement is that the elements in each source share some value that can be compared for equality. For example, a food distributor might have a list of suppliers of a certain product, and a list of buyers. A `join` clause can be used, for example, to create a list of the suppliers and buyers of that product who are all in the same specified region.

A `join` clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The `join` clause compares the specified keys for equality by using the special `equals` keyword.

The following example shows a simple inner join. This query produces a flat sequence of "product name / category" pairs. The same category string will appear

in multiple elements. If an element from `categories` has no matching `products`, that category will not appear in the results.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

8.2.5 group clause

The `group` clause returns a sequence of IGrouping<TKey,TElement> objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type char, and is stored in the `Key` property of each IGrouping<TKey,TElement> object. The compiler infers the type of the key.

You can end a query expression with a `group` clause, as shown in the following example:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Returns a list of groups of students. In each group of students the last name has the same first letter.

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the into contextual keyword. When you use `into`, you must continue with the query, and eventually end it with either a `select` statement or another `group` clause, as shown in the following excerpt:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

The difference of the query above with the first query is that the groups of students are now sorted on the first letter of the last name of the students.

8.2.5.1 Enumerating the results of a group query

Because the IGrouping<TKey,TElement> objects produced by a `group` query are essentially a list of lists, you must use a nested foreach loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each item in the group itself. A group may have a key but no elements. The following is the `foreach` loop that executes the query in the previous code examples:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

8.2.5.2 Key types

Group keys can be any type, such as a string, a built-in numeric type, or a user-defined named type or anonymous type.

A composite key can be used when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element. In the following example, assume that a class `Person` has been declared with members named `surname` and `city`. The `group` clause causes a separate group to be created for each set of persons with the same last name and the same city.

```
group person by new {name = person.surname, city = person.city};
```

8.2.6 orderby clause

In a query expression, the `orderby` clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending.

In the following example, the first query sorts the words in alphabetical order starting from A, and second query sorts the same words in descending order. (The `ascending` keyword is the default sort value and can be omitted.)

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

8.2.6 let clause

In a query expression, it is sometimes useful to store the result of a sub-expression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply. Once initialized with a value, the range variable cannot be used to store another value. However, if the range variable holds a queryable type, it can be queried.

In the following example `let` is used in two ways:

1. To create an enumerable type that can itself be queried.
2. To enable the query to call `ToLower` only one time on the range variable `word`. Without using `let`, you would have to call `ToLower` in each predicate in the `where` clause.

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

8.3 Method Syntax

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You can call them directly by using method syntax instead of query syntax.

Query syntax and method syntax are semantically identical, but many people find query syntax simpler and easier to read. Some queries must be expressed as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. The reference documentation for the standard query operators in the System.Linq namespace generally uses method syntax. Therefore, even when getting started writing LINQ queries, it is useful to be familiar with how to use method syntax in queries and in query expressions themselves.

8.3.1 Standard Query Operator Extension Methods

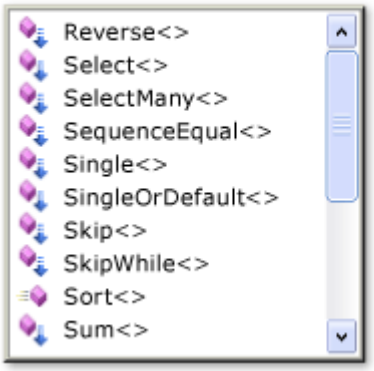
The following example shows a simple *query expression* and the semantically equivalent query written as a *method-based query*.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

The output from the two examples is identical. You can see that the type of the query variable is the same in both forms: IEnumerable<T>.

To understand the method-based query, let's examine it more closely. On the right side of the expression, notice that the `where` clause is now expressed as an instance method on the `numbers` object, which as you will recall has a type of `IEnumerable<int>`. If you are familiar with the generic IEnumerable<T> interface, you know that it does not have a `Where` method. However, if you invoke the IntelliSense completion list in the Visual Studio IDE, you will see not only a `Where` method, but many other methods such as `Select`, `SelectMany`, `Join`, and `Orderby`. These are all the standard query operators.

```
List<string> list = new List<string();
list.
```



Although it looks as if IEnumerable<T> has been redefined to include these additional methods, in fact this is not the case. The standard query operators are implemented as a new kind of method called *extension methods*. Extensions methods "extend" an existing type; they can be called as if they were instance methods on the type. The standard query operators extend IEnumerable<T> and that is why you can write `numbers.Where(...)`.

To get started using LINQ, all that you really have to know about extension methods is how to bring them into scope in your application by using the correct `using` directives. From your application's point of view, an extension method and a regular instance method are the same.

8.3.2 Lambda Expressions

In the previous example, notice that the conditional expression (`num % 2 == 0`) is passed as an in-line argument to the `Where` method: `Where(num => num % 2 == 0)`. This inline expression is called a lambda expression. It is a convenient way to write code that would otherwise have to be written in more cumbersome form as an anonymous method or a generic delegate or an expression tree. In C# `=>` is the lambda operator, which is read as "goes to". The `num` on the left of the operator is the input variable which corresponds to `num` in the query expression. The compiler can infer the type of `num` because it knows that `numbers` is a generic IEnumerable<T> type. The body of the lambda is just the same as the expression in query syntax or in any other C# expression or statement; it can include method calls and other complex logic. The "return value" is just the expression result.

To get started using LINQ, you do not have to use lambdas extensively. However, certain queries can only be expressed in method syntax and some of those require lambda expressions. After you become more familiar with lambdas, you will find that they are a powerful and flexible tool in your LINQ toolbox.

8.3.3 Composability of Queries

In the previous code example, note that the `OrderBy` method is invoked by using the dot operator on the call to `Where`. `Where` produces a filtered sequence, and then `Orderby` operates on that sequence by sorting it. Because queries return an `IEnumerable`, you compose them in method syntax by chaining the method calls together. This is what the compiler does behind the scenes when you write queries by using query syntax. And because a query variable does not store the results of the query, you can modify it or use it as the basis for a new query at any time, even after it has been executed.

8.4 Sources

[1] Microsoft, “C# Documentation, Language Integrated Query”, Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq

[2] Microsoft, “C# Documentation, Query keywords”, Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords