Table of Contents

Language Integrated Query (LINQ)

1/5/2018 • 3 min to read • Edit Online

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.

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 use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

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

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see Type relationships in LINQ query operations.
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see Introduction to LINQ queries.

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see C# language specification and Standard query operators overview.
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see Query syntax and method syntax in LINQ.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in Query expression basics, and then read the documentation for the LINQ technology in which you are interested:

- XML documents: LINQ to XML
- ADO.NET Entity Framework: LINQ to entities
- .NET collections, files, strings and so on: LINQ to objects

To gain a deeper understanding of LINQ in general, see LINQ in C#.

To start working with LINQ in C#, see the tutorial Working with LINQ.

Query expression basics

1/5/2018 • 12 min to read • Edit Online

What is a query and what does it do?

A *query* is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.

Generally, the source data is organized logically as a sequence of elements of the same kind. For example, a SQL database table contains a sequence of rows. In an XML file, there is a "sequence" of XML elements (although these are organized hierarchically in a tree structure). An in-memory collection contains a sequence of objects.

From an application's viewpoint, the specific type and structure of the original source data is not important. The application always sees the source data as an IEnumerable<T> or IQueryable<T> collection. For example, in LINQ to XML, the source data is made visible as an `IEnumerable` <XElement>.

Given this source sequence, a query may do one of three things:

- Retrieve a subset of the elements to produce a new sequence without modifying the individual elements. The query may then sort or group the returned sequence in various ways, as shown in the following example (assume `scores` is an `int[]`):

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Retrieve a sequence of elements as in the previous example but transform them to a new type of object. For example, a query may retrieve only the last names from certain customer records in a data source. Or it may retrieve the complete record and then use it to construct another in-memory object type or even XML data before generating the final result sequence. The following example shows a projection from an `int` to a `string`. Note the new type of `highScoresQuery`.

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select String.Format("The score is {0}", score);
  ```

- Retrieve a singleton value about the source data, such as:

  - The number of elements that match a certain condition.

  - The element that has the greatest or least value.

  - The first element that matches a condition, or the sum of particular values in a specified set of elements. For example, the following query returns the number of scores greater than 80 from the `scores` integer array:

```
int highScoreCount =
    (from score in scores
     where score > 80
     select score)
     .Count();
```

In the previous example, note the use of parentheses around the query expression before the call to the `Count` method. You can also express this by using a new variable to store the concrete result. This technique is more readable because it keeps the variable that stores the query separate from the query that stores a result.

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

In the previous example, the query is executed in the call to `Count`, because `Count` must iterate over the results in order to determine the number of elements returned by `highScoresQuery`.

What is a query expression?

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 or XQuery. 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.

Query variable

In LINQ, a query variable is any variable that stores a *query* instead of the *results* of a query. More specifically, a query variable is always an enumerable type that will produce a sequence of elements when it is iterated over in a `foreach` statement or a direct call to its `IEnumerator.MoveNext` method.

The following code example shows a simple query expression with one data source, one filtering clause, one ordering clause, and no transformation of the source elements. The `select` clause ends the query.

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

In the previous example, `scoreQuery` is a *query variable,* which is sometimes referred to as just a *query*. The query variable stores no actual result data, which is produced in the `foreach` loop. And when the `foreach` statement executes, the query results are not returned through the query variable `scoreQuery`. Rather, they are returned through the iteration variable `testScore`. The `scoreQuery` variable can be iterated in a second `foreach` loop. It will produce the same results as long as neither it nor the data source has been modified.

A query variable may store a query that is expressed in query syntax or method syntax, or a combination of the two. In the following examples, both `queryMajorCities` and `queryMajorCities2` are query variables:

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;


// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);
```

On the other hand, the following two examples show variables that are not query variables even though each is initialized with a query. They are not query variables because they store results:

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

For more information about the different ways to express queries, see Query syntax and method syntax in LINQ.

Explicit and implicit typing of query variables

This documentation usually provides the explicit type of the query variable in order to show the type relationship between the query variable and the select clause. However, you can also use the var keyword to instruct the compiler to infer the type of a query variable (or any other local variable) at compile time. For example, the query example that was shown previously in this topic can also be expressed by using implicit typing:

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

For more information, see Implicitly typed local variables and Type relationships in LINQ query operations.

Starting a query expression

A query expression must begin with a `from` clause. It specifies a data source together with a range variable. The range variable represents each successive element in the source sequence as the source sequence is being traversed. The range variable is strongly typed based on the type of elements in the data source. In the following example, because `countries` is an array of `Country` objects, the range variable is also typed as `Country`. Because the range variable is strongly typed, you can use the dot operator to access any available members of the type.

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

The range variable is in scope until the query is exited either with a semicolon or with a *continuation* clause.

A query expression may contain multiple `from` clauses. Use additional `from` clauses when each element in the source sequence is itself a collection or contains a collection. For example, assume that you have a collection of `Country` objects, each of which contains a collection of `City` objects named `Cities`. To query the `City` objects in each `Country`, use two `from` clauses as shown here:

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

For more information, see from clause.

Ending a query expression

A query expression must end with either a `group` clause or a `select` clause.

group clause

Use the `group` clause to produce a sequence of groups organized by a key that you specify. The key can be any data type. For example, the following query creates a sequence of groups that contains one or more `Country` objects and whose key is a `char` value.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

For more information about grouping, see group clause.

select clause

Use the `select` clause to produce all other types of sequences. A simple `select` clause just produces a sequence of the same type of objects as the objects that are contained in the data source. In this example, the data source contains `Country` objects. The `orderby` clause just sorts the elements into a new order and the `select` clause produces a sequence of the reordered `Country` objects.

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

The `select` clause can be used to transform source data into sequences of new types. This transformation is also named a *projection*. In the following example, the `select` clause *projects* a sequence of anonymous types which contains only a subset of the fields in the original element. Note that the new objects are initialized by using an object initializer.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

For more information about all the ways that a `select` clause can be used to transform source data, see select clause.

Continuations with "into"

You can use the `into` keyword in a `select` or `group` clause to create a temporary identifier that stores a query. Do this when you must perform additional query operations on a query after a grouping or select operation. In the following example `countries` are grouped according to population in ranges of 10 million. After these groups are created, additional clauses filter out some groups, and then to sort the groups in ascending order. To perform those additional operations, the continuation represented by `countryGroup` is required.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

For more information, see into.

Filtering, ordering, and joining

Between the starting `from` clause, and the ending `select` or `group` clause, all other clauses (`where`, `join`, `orderby`, `from`, `let`) are optional. Any of the optional clauses may be used zero times or multiple times in a query body.

where clause

Use the `where` clause to filter out elements from the source data based on one or more predicate expressions. The `where` clause in the following example has one predicate with two conditions.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

For more information, see where clause.

orderby clause

Use the `orderby` clause to sort the results in either ascending or descending order. You can also specify secondary sort orders. The following example performs a primary sort on the `country` objects by using the `Area` property. It then performs a secondary sort by using the `Population` property.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

The `ascending` keyword is optional; it is the default sort order if no order is specified. For more information, see orderby clause.

join clause

Use the `join` clause to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element. In LINQ, join operations are performed on sequences of objects whose elements are different types. After you have joined two sequences, you must use a `select` or `group` statement to specify which element to store in the output sequence. You can also use an anonymous type to combine properties from each set of associated elements into a new type for the output sequence. The following example associates `prod` objects whose `Category` property matches one of the categories in the `categories` string array. Products whose `Category` does not match any string in `categories` are filtered out. The `select` statement projects a new type whose properties are taken from both `cat` and `prod`.

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

You can also perform a group join by storing the results of the `join` operation into a temporary variable by using the into keyword. For more information, see join clause.

let clause

Use the `let` clause to store the result of an expression, such as a method call, in a new range variable. In the following example, the range variable `firstName` stores the first element of the array of strings that is returned by `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

For more information, see let clause.

Subqueries in a query expression

A query clause may itself contain a query expression, which is sometimes referred to as a *subquery*. Each subquery starts with its own `from` clause that does not necessarily point to the same data source in the first `from` clause. For example, the following query shows a query expression that is used in the select statement to retrieve the results of a grouping operation.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

For more information, see How to: perform a subquery on a grouping operation.

See Also

C# programming guide
LINQ query expressions

Query keywords (LINQ)

Standard query operators overview

LINQ in C#

1/5/2018 • 1 min to read • Edit Online

This section contains links to topics that provide more detailed information about LINQ.

In this section

Introduction to LINQ queries
Describes the three parts of the basic LINQ query operation that are common across all languages and data sources.

LINQ and generic types
Provides a brief introduction to generic types as they are used in LINQ.

Data transformations with LINQ
Describes the various ways that you can transform data retrieved in queries.

Type relationships in LINQ query operations
Describes how types are preserved and/or transformed in the three parts of a LINQ query operation

Query syntax and method syntax in LINQ
Compares method syntax and query syntax as two ways to express a LINQ query.

C# features that support LINQ
Describes the language constructs in C# that support LINQ.

Related sections

LINQ query expressions
Includes an overview of queries in LINQ and provides links to additional resources.

Standard query operators overview
Introduces the standard methods used in LINQ.

Write LINQ queries in C#

1/5/2018 • 4 min to read • Edit Online

This topic shows the three ways in which you can write a LINQ query in C#:

1. Use query syntax.
2. Use method syntax.
3. Use a combination of query syntax and method syntax.

The following examples demonstrate some simple LINQ queries by using each approach listed previously. In general, the rule is to use (1) whenever possible, and use (2) and (3) whenever necessary.

NOTE

These queries operate on simple in-memory collections; however, the basic syntax is identical to that used in LINQ to Entities and LINQ to XML.

Example

Query syntax

The recommended way to write most queries is to use *query syntax* to create *query expressions*. The following example shows three query expressions. The first query expression demonstrates how to filter or restrict results by applying conditions with a `where` clause. It returns all elements in the source sequence whose values are greater than 7 or less than 3. The second expression demonstrates how to order the returned results. The third expression demonstrates how to group results according to a key. This query returns two groups based on the first letter of the word.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Note that the type of the queries is IEnumerable<T>. All of these queries could be written using `var` as shown in the following example:

`var query = from num in numbers...`

In each previous example, the queries do not actually execute until you iterate over the query variable in a `foreach` statement or other statement. For more information, see Introduction to LINQ Queries.

Example

Method syntax

Some query operations must be expressed as a method call. The most common such methods are those that return singleton numeric values, such as Sum, Max, Min, Average, and so on. These methods must always be called last in any query because they represent only a single value and cannot serve as the source for an additional query operation. The following example shows a method call in a query expression:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Example

If the method has Action or Func parameters, these are provided in the form of a lambda expression, as shown in the following example:

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

In the previous queries, only Query #4 executes immediately. This is because it returns a single value, and not a generic IEnumerable<T> collection. The method itself has to use `foreach` in order to compute its value.

Each of the previous queries can be written by using implicit typing with var, as shown in the following example:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Example

Mixed query and method syntax

This example shows how to use method syntax on the results of a query clause. Just enclose the query expression in parentheses, and then apply the dot operator and call the method. In the following example, query #7 returns a count of the numbers whose value is between 3 and 7. In general, however, it is better to use a second variable to store the result of the method call. In this manner, the query is less likely to be confused with the results of the query.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Because Query #7 returns a single value and not a collection, the query executes immediately.

The previous query can be written by using implicit typing with `var`, as follows:

```
var numCount = (from num in numbers...
```

It can be written in method syntax as follows:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

It can be written by using explicit typing, as follows:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

See Also

Walkthrough: Writing Queries in C#
LINQ Query Expressions
where clause

Query a collection of objects

1/5/2018 • 2 min to read • Edit Online

This example shows how to perform a simple query over a list of `Student` objects. Each `Student` object contains some basic information about the student, and a list that represents the student's scores on four examinations.

This application serves as the framework for many other examples in this section that use the same `students` data source.

Example

The following query returns the students who received a score of 90 or greater on their first exam.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }



    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

This query is intentionally simple to enable you to experiment. For example, you can try more conditions in the `where` clause, or use an `orderby` clause to sort the results.

See also

LINQ Query Expressions
Interpolated Strings

How to: Return a Query from a Method (C# Programming Guide)

1/5/2018 • 2 min to read • Edit Online

This example shows how to return a query from a method as the return value and as an `out` parameter.

Query objects are composable, meaning that you can return a query from a method. Objects that represent queries do not store the resulting collection, but rather the steps to produce the results when needed. The advantage of returning query objects from methods is that they can be further composed or modified. Therefore any return value or `out` parameter of a method that returns a query must also have that type. If a method materializes a query into a concrete List<T> or Array type, it is considered to be returning the query results instead of the query itself. A query variable that is returned from a method can still be composed or modified.

Example

In the following example, the first method returns a query as a return value, and the second method returns a query as an `out` parameter. Note that in both cases it is a query that is returned, not query results.

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }


        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out parameter.
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }


        // You can modify a query by using query composition. A saved query
        // is nested inside a new query definition that revises the results
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1:");
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

See Also

LINQ Query Expressions

Store the results of a query in memory

1/5/2018 • 1 min to read • Edit Online

A query is basically a set of instructions for how to retrieve and organize data. Queries are executed lazily, as each subsequent item in the result is requested. When you use `foreach` to iterate the results, items are returned as accessed. To evaluate a query and store its results without executing a `foreach` loop, just call one of the following methods on the query variable:

- ToList
- ToArray
- ToDictionary
- ToLookup

We recommend that when you store the query results, you assign the returned collection object to a new variable as shown in the following example:

Example

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

See Also

LINQ Query Expressions

Group query results

1/5/2018 • 8 min to read • Edit Online

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

- By a single property.
- By the first letter of a string property.
- By a computed numeric range.
- By Boolean predicate or other expression.
- By a compound key.

In addition, the last two queries project their results into a new anonymous type that contains only the student's first and last name. For more information, see the group clause.

Example

All the examples in this topic use the following helper classes and data sources.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }



    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Example

The following example shows how to group source elements by using a single property of the element as the group key. In this case the key is a `string`, the student's last name. It is also possible to use a substring for the key. The grouping operation uses the default equality comparer for the type.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySingleProperty()`.

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine("Key: {0}", nameGroup.Key);
        foreach (var student in nameGroup)
        {
            Console.WriteLine("\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Example

The following example shows how to group source elements by using something other than a property of the object for the group key. In this example, the key is the first letter of the student's last name.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySubstring()`.

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine("\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Example

The following example shows how to group source elements by using a numeric range as a group key. The query then projects the results into an anonymous type that contains only the first and last name and the percentile range to which the student belongs. An anonymous type is used because it is not necessary to use the complete `Student` object to display the results. `GetPercentile` is a helper function that calculates a percentile based on the student's average score. The method returns an integer between 0 and 10.

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByRange()`.

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine("\t{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Example

The following example shows how to group source elements by using a Boolean comparison expression. In this example, the Boolean expression tests whether a student's average exam score is greater than 75. As in previous examples, the results are projected into an anonymous type because the complete source element is not needed. Note that the properties in the anonymous type become properties on the `Key` member and can be accessed by name when the query is executed.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByBoolean()`.

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Example

The following example shows how to use an anonymous type to encapsulate a key that contains multiple values. In this example, the first key value is the first letter of the student's last name. The second key value is a Boolean that specifies whether the student scored over 85 on the first exam. You can order the groups by any property in the key.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByCompositeKey()`.

```
public void GroupByCompositeKey()
{

    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        ($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"\t{item.FirstName} {item.LastName}");
        }
    }
}
/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

See also

GroupBy
IGrouping<TKey,TElement>
LINQ Query Expressions
group clause
Anonymous Types
Perform a Subquery on a Grouping Operation
Create a Nested Group
Grouping Data

Create a nested group

1/5/2018 • 1 min to read • Edit Online

The following example shows how to create nested groups in a LINQ query expression. Each group that is created according to student year or grade level is then further subdivided into groups based on the individuals' names.

Example

NOTE

This example contains references to objects that are defined in the sample code in Query a collection of objects.

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine("\t\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

Note that three nested `foreach` loops are required to iterate over the inner elements of a nested group.

See also

LINQ Query Expressions

Perform a subquery on a grouping operation

1/5/2018 • 1 min to read • Edit Online

This topic shows two different ways to create a query that orders the source data into groups, and then performs a subquery over each group individually. The basic technique in each example is to group the source elements by using a *continuation* named `newGroup`, and then generating a new subquery against `newGroup`. This subquery is run against each new group that is created by the outer query. Note that in this particular example the final output is not a group, but a flat sequence of anonymous types.

For more information about how to group, see group clause.

For more information about continuations, see into. The following example uses an in-memory data structure as the data source, but the same principles apply for any kind of LINQ data source.

Example

NOTE

This example contains references to objects that are defined in the sample code in Query a collection of objects.

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

See also

LINQ Query Expressions

Group results by contiguous keys

1/5/2018 • 7 min to read • Edit Online

The following example shows how to group elements into chunks that represent subsequences of contiguous keys. For example, assume that you are given the following sequence of key-value pairs:

KEY	VALUE
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

The following groups will be created in this order:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

The solution is implemented as an extension method that is thread-safe and that returns its results in a streaming manner. In other words, it produces its groups as it moves through the source sequence. Unlike the `group` or `orderby` operators, it can begin returning groups to the caller before all of the sequence has been read.

Thread-safety is accomplished by making a copy of each group or chunk as the source sequence is iterated, as explained in the source code comments. If the source sequence has a large sequence of contiguous items, the common language runtime may throw an OutOfMemoryException.

Example

The following example shows both the extension method and the client code that uses it.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

            // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk. Each
```

```
ChunkItem
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }
            // The value that is used to determine matching elements
            private readonly TKey key;

            // Stores a reference to the enumerator for the source sequence
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare keys.
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the end of the source sequence.
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the list contains > 1 elements.
                tail = head;

                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied to the list of ChunkItems,
            // and the source enumerator is either at the end, or else on an element with a new key.
            // the tail of the linked list is set to null in the CopyNextChunkElement method if the
            // key of the next element does not match the current chunk's key, or there are no more elements in
the source.
            private bool DoneCopyingChunk => tail == null;

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequence.
                // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) at the end of the current chunk
                // then null out the enumerator and predicate for reuse with the next chunk.
                if (isLastSourceElement || !predicate(enumerator.Current))
                {
```

```
                enumerator = null;
                predicate = null;
            }
            else
            {
                tail.Next = new ChunkItem(enumerator.Current);
            }

            // tail will be null if we are at the end of the chunk elements
            // This check is made in DoneCopyingChunk.
            tail = tail.Next;
        }

        // Called after the end of the last chunk was reached. It first checks whether
        // there are more elements in the source sequence. If there are, it
        // Returns true if enumerator for this chunk was exhausted.
        internal bool CopyAllChunkElements()
        {
            while (true)
            {
                lock (m_Lock)
                {
                    if (DoneCopyingChunk)
                    {
                        // If isLastSourceElement is false,
                        // it signals to the outer iterator
                        // to continue iterating.
                        return isLastSourceElement;
                    }
                    else
                    {
                        CopyNextChunkElement();
                    }
                }
            }
        }

        public TKey Key => key;

        // Invoked by the inner foreach loop. This method stays just one step ahead
        // of the client requests. It adds the next element of the chunk only after
        // the clients requests the last element in the list so far.
        public IEnumerator<TSource> GetEnumerator()
        {
            //Specify the initial element to enumerate.
            ChunkItem current = head;

            // There should always be at least one ChunkItem in a Chunk.
            while (current != null)
            {
                // Yield the current item in the list.
                yield return current.Value;

                // Copy the next item from the source sequence,
                // if we are at the end of our local list.
                lock (m_Lock)
                {
                    if (current == tail)
                    {
                        CopyNextChunkElement();
                    }
                }

                // Move to the next ChunkItem in the list.
                current = current.Next;
            }
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()  => GetEnumerator();
```

```
            {
                return GetEnumerator();
            }
        }
    }

    // A simple named type is used for easier viewing in the debugger. Anonymous types
    // work just as well with the ChunkBy operator.
    public class KeyValPair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    class Program
    {
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;


        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initializer.
            list = new[]
        {
            new KeyValPair{ Key = "A", Value = "We" },
            new KeyValPair{ Key = "A", Value = "Think" },
            new KeyValPair{ Key = "A", Value = "That" },
            new KeyValPair{ Key = "B", Value = "Linq" },
            new KeyValPair{ Key = "C", Value = "Is" },
            new KeyValPair{ Key = "A", Value = "Really" },
            new KeyValPair{ Key = "B", Value = "Cool" },
            new KeyValPair{ Key = "B", Value = "!" }
        };

            // Create the query by using our user-defined query operator.
            query = list.ChunkBy(p => p.Key);


            // ChunkBy returns IGrouping objects, therefore a nested
            // foreach loop is required to access the elements in each "chunk".
            foreach (var item in query)
            {
                Console.WriteLine("Group key = {0}", item.Key);
                foreach (var inner in item)
                {
                    Console.WriteLine($"\t{inner.Value}");
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

     }
}
```

To use the extension method in your project, copy the `MyExtensions` static class to a new or existing source code file and if it is required, add a `using` directive for the namespace where it is located.

See also

Dynamically specify predicate filters at runtime

1/5/2018 • 2 min to read • Edit Online

In some cases you do not know until run time how many predicates you have to apply to source elements in the `where` clause. One way to dynamically specify multiple predicate filters is to use the Contains method, as shown in the following example. The example is constructed in two ways. First, the project is run by filtering on values that are provided in the program. Then the project is run again by using input provided at run time.

To filter by using the Contains method

1. Open a new console application and name it `PredicateFilters`.
2. Copy the `StudentClass` class from Query a collection of objects and paste it into namespace `PredicateFilters` underneath class `Program`. `StudentClass` provides a list of `Student` objects.
3. Comment out the `Main` method in `StudentClass`.
4. Replace class `Program` with the following code.

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Add the following line to the `Main` method in class `DynamicPredicates`, under the declaration of `ids`.

```
QueryById(ids);
```

6. Run the project.
7. The following output is displayed in a console window:

   Garcia: 114

   O'Donnell: 112

Omelchenko: 111

8. The next step is to run the project again, this time by using input entered at run time instead of array `ids`. Change `QueryByID(ids)` to `QueryByID(args)` in the `Main` method.

9. Run the project with the command line arguments `122 117 120 115`. When the project is run, those values become elements of `args`, the parameter of the `Main` method..

10. The following output is displayed in a console window:

    Adams: 120

    Feng: 117

    Garcia: 115

    Tucker: 122

To filter by using a switch statement

1. You can use a `switch` statement to select among predetermined alternative queries. In the following example, `studentQuery` uses a different `where` clause depending on which grade level, or year, is specified at run time.

2. Copy the following method and paste it into class `DynamicPredicates`.

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine("The following students are at level {0}", year.ToString());
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. In the `Main` method, replace the call to `QueryByID` with the following call, which sends the first element from the `args` array as its argument: `QueryByYear(args[0])`.

4. Run the project with a command line argument of an integer value between 1 and 4.

See Also

LINQ Query Expressions
where clause

Perform inner joins

1/5/2018 • 10 min to read • Edit Online

In relational database terms, an *inner join* produces a result set in which each element of the first collection appears one time for every matching element in the second collection. If an element in the first collection has no matching elements, it does not appear in the result set. The Join method, which is called by the `join` clause in C#, implements an inner join.

This topic shows you how to perform four variations of an inner join:

- A simple inner join that correlates elements from two data sources based on a simple key.
- An inner join that correlates elements from two data sources based on a *composite* key. A composite key, which is a key that consists of more than one value, enables you to correlate elements based on more than one property.
- A *multiple join* in which successive join operations are appended to each other.
- An inner join that is implemented by using a group join.

Example

Simple key join example

The following example creates two collections that contain objects of two user-defined types, `Person` and `Pet`. The query uses the `join` clause in C# to match `Person` objects with `Pet` objects whose `Owner` is that `Person`. The `select` clause in C# defines how the resulting objects will look. In this example the resulting objects are anonymous types that consist of the owner's first name and the pet's name.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui
```

Note that the `Person` object whose `LastName` is "Huff" does not appear in the result set because there is no `Pet` object that has `Pet.Owner` equal to that `Person`.

Example

Composite key join example

Instead of correlating elements based on just one property, you can use a composite key to compare elements based on multiple properties. To do this, specify the key selector function for each collection to return an anonymous type that consists of the properties you want to compare. If you label the properties, they must have the same label in each key's anonymous type. The properties must also appear in the same order.

The following example uses a list of `Employee` objects and a list of `Student` objects to determine which employees are also students. Both of these types have a `FirstName` and a `LastName` property of type String. The functions that create the join keys from each list's elements return an anonymous type that consists of the `FirstName` and `LastName` properties of each element. The join operation compares these composite keys for equality and returns pairs of objects from each list where both the first name and the last name match.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
         new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
         new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
         new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Example

Multiple join example

Any number of join operations can be appended to each other to perform a multiple join. Each `join` clause in C# correlates a specified data source with the results of the previous join.

The following example creates three collections: a list of `Person` objects, a list of `Cat` objects, and a list of `Dog` objects.

The first `join` clause in C# matches people and cats based on a `Person` object matching `Cat.Owner`. It returns a sequence of anonymous types that contain the `Person` object and `Cat.Name`.

The second `join` clause in C# correlates the anonymous types returned by the first join with `Dog` objects in the supplied list of dogs, based on a composite key that consists of the `Owner` property of type `Person`, and the first letter of the animal's name. It returns a sequence of anonymous types that contain the `Cat.Name` and `Dog.Name` properties from each matching pair. Because this is an inner join, only those objects from the first data source that have a match in the second data source are returned.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name,
            with \"{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".
```

Example

Inner join by using grouped join example

The following example shows you how to implement an inner join by using a group join.

In `query1`, the list of `Person` objects is group-joined to the list of `Pet` objects based on the `Person` matching the `Pet.Owner` property. The group join creates a collection of intermediate groups, where each group consists of a `Person` object and a sequence of matching `Pet` objects.

By adding a second `from` clause to the query, this sequence of sequences is combined (or flattened) into one longer sequence. The type of the elements of the final sequence is specified by the `select` clause. In this example, that type is an anonymous type that consists of the `Person.FirstName` and `Pet.Name` properties for each matching pair.

The result of `query1` is equivalent to the result set that would have been obtained by using the `join` clause without the `into` clause to perform an inner join. The `query2` variable demonstrates this equivalent query.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into gj
                 from subpet in gj
                 select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName = pet.Name };

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

See also

[Join](#)
[GroupJoin](#)
[Perform grouped joins](#)
[Perform left outer joins](#)
[Anonymous types](#)

Perform grouped joins

1/5/2018 • 4 min to read • Edit Online

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection.

For example, a class or a relational database table named `Student` might contain two fields: `Id` and `Name`. A second class or relational database table named `Course` might contain two fields: `StudentId` and `CourseTitle`. A group join of these two data sources, based on matching `Student.Id` and `Course.StudentId`, would group each `Student` with a collection of `Course` objects (which might be empty).

NOTE

Each element of the first collection appears in the result set of a group join regardless of whether correlated elements are found in the second collection. In the case where no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This differs from the result selector in a non-group join, which cannot access elements from the first collection that have no match in the second collection.

The first example in this topic shows you how to perform a group join. The second example shows you how to use a group join to create XML elements.

Example

Group join example

The following example performs a group join of objects of type `Person` and `Pet` based on the `Person` matching the `Pet.Owner` property. Unlike a non-group join, which would produce a pair of elements for each match, the group join produces only one resulting object for each element of the first collection, which in this example is a `Person` object. The corresponding elements from the second collection, which in this example are `Pet` objects, are grouped into a collection. Finally, the result selector function creates an anonymous type for each match that consists of `Person.FirstName` and a collection of `Pet` objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine("{0}:", v.OwnerName);
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine("  {0}", pet.Name);
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

Example

Group join to create XML example

Group joins are ideal for creating XML by using LINQ to XML. The following example is similar to the previous example except that instead of creating anonymous types, the result selector function creates XML elements that represent the joined objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

See also

Join

GroupJoin

Perform inner joins

Perform left outer joins

Anonymous types

Perform left outer joins

1/5/2018 • 2 min to read • Edit Online

A left outer join is a join in which each element of the first collection is returned, regardless of whether it has any correlated elements in the second collection. You can use LINQ to perform a left outer join by calling the DefaultIfEmpty method on the results of a group join.

Example

The following example demonstrates how to use the DefaultIfEmpty method on the results of a group join to perform a left outer join.

The first step in producing a left outer join of two collections is to perform an inner join by using a group join. (See Perform inner joins for an explanation of this process.) In this example, the list of `Person` objects is inner-joined to the list of `Pet` objects based on a `Person` object that matches `Pet.Owner`.

The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection. This is accomplished by calling DefaultIfEmpty on each sequence of matching elements from the group join. In this example, DefaultIfEmpty is called on each sequence of matching `Pet` objects. The method returns a collection that contains a single, default value if the sequence of matching `Pet` objects is empty for any `Person` object, thereby ensuring that each `Person` object is represented in the result collection.

NOTE

The default value for a reference type is `null`; therefore, the example checks for a null reference before accessing each element of each `Pet` collection.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

See also

Join

GroupJoin

Perform inner joins

Perform grouped joins

Anonymous types

Order the results of a join clause

1/5/2018 • 1 min to read • Edit Online

This example shows how to order the results of a join operation. Note that the ordering is performed after the join. Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we do not recommend it. Some LINQ providers might not preserve that ordering after the join.

Example

This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a sub-query orders all the matching elements from the products sequence.

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
  {
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
  };

        // Specify the second data source.
        List<Product> products = new List<Product>()
 {
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
  };
        #endregion
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();

            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();

        }
        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.CategoryID into prodGroup
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine("  {prodItem.Name:-10} {prodItem.CategoryID}");
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

See also

[LINQ query expressions]

[orderby clause]

[join clause]

Join by using composite keys

1/5/2018 • 1 min to read • Edit Online

This example shows how to perform join operations in which you want to use more than one key to define a match. This is accomplished by using a composite key. You create a composite key as an anonymous type or named typed with the values that you want to compare. If the query variable will be passed across method boundaries, use a named type that overrides Equals and GetHashCode for the key. The names of the properties, and the order in which they occur, must be identical in each key.

Example

The following example demonstrates how to use a composite key to join data from three tables:

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID,        d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

Type inference on composite keys depends on the names of the properties in the keys, and the order in which they occur. If the properties in the source sequences do not have the same names, you must assign new names in the keys. For example, if the `Orders` table and `OrderDetails` table each used different names for their columns, you could create composite keys by assigning identical names in the anonymous types:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Composite keys can be also used in a `group` clause.

See also

LINQ query expressions
join clause
group clause

Perform custom join operations

1/5/2018 • 5 min to read • Edit Online

This example shows how to perform join operations that are not possible with the `join` clause. In a query expression, the `join` clause is limited to, and optimized for, equijoins, which are by far the most common type of join operation. When performing an equijoin, you will probably always get the best performance by using the `join` clause.

However, the `join` clause cannot be used in the following cases:

- When the join is predicated on an expression of inequality (a non-equijoin).
- When the join is predicated on more than one expression of equality or inequality.
- When you have to introduce a temporary range variable for the right side (inner) sequence before the join operation.

To perform joins that are not equijoins, you can use multiple `from` clauses to introduce each data source independently. You then apply a predicate expression in a `where` clause to the range variable for each source. The expression also can take the form of a method call.

> **NOTE**
>
> Do not confuse this kind of custom join operation with the use of multiple `from` clauses to access inner collections. For more information, see join clause.

Example

The first method in the following example shows a simple cross join. Cross joins must be used with caution because they can produce very large result sets. However, they can be useful in some scenarios for creating source sequences against which additional queries are run.

The second method produces a sequence of all the products whose category ID is listed in the category list on the left side. Note the use of the `let` clause and the `Contains` method to create a temporary array. It also is possible to create the array before the query and eliminate the first `from` clause.

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
    };

            // Specify the second data source.
            List<Product> products = new List<Product>()
   {
      new Product{Name="Tea",  CategoryID=001},
      new Product{Name="Mustard", CategoryID=002},
      new Product{Name="Pickles", CategoryID=002},
      new Product{Name="Carrots", CategoryID=003},
      new Product{Name="Bok Choy", CategoryID=003},
      new Product{Name="Peaches", CategoryID=005},
      new Product{Name="Melons", CategoryID=005},
      new Product{Name="Ice Cream", CategoryID=007},
      new Product{Name="Mackerel", CategoryID=012},
    };
            #endregion

            static void Main()
            {
                CustomJoins app = new CustomJoins();
                app.CrossJoin();
                app.NonEquijoin();

                Console.WriteLine("Press any key to exit.");
                Console.ReadKey();
            }

            void CrossJoin()
            {
                var crossJoinQuery =
                    from c in categories
                    from p in products
                    select new { c.ID, p.Name };

                Console.WriteLine("Cross Join Query:");
                foreach (var v in crossJoinQuery)
                {
                    Console.WriteLine($"{v.ID:-5}{v.Name}");
                }
            }

            void NonEquijoin()
            {
                var nonEquijoinQuery =
                    from p in products
                    let catIds = from c in categories
                                 select c.ID
                    where catIds.Contains(p.CategoryID) == true
                    select new { Product = p.Name, CategoryID = p.CategoryID };

                Console.WriteLine("Non-equijoin query:");
                foreach (var v in nonEquijoinQuery)
                {
                    Console.WriteLine($"{v.CategoryID:-5}{v.Product}");
                }
            }
        }
        /* Output:
    Cross Join Query:
    1    Tea
    1    Mustard
    1    Pickles
    1    Carrots
    1    Bok Choy
    1    Peaches
```

```
1    Melons
1    Ice Cream
1    Mackerel
2    Tea
2    Mustard
2    Pickles
2    Carrots
2    Bok Choy
2    Peaches
2    Melons
2    Ice Cream
2    Mackerel
3    Tea
3    Mustard
3    Pickles
3    Carrots
3    Bok Choy
3    Peaches
3    Melons
3    Ice Cream
3    Mackerel
Non-equijoin query:
1    Tea
2    Mustard
2    Pickles
3    Carrots
3    Bok Choy
Press any key to exit.
     */
```

Example

In the following example, the query must join two sequences based on matching keys that, in the case of the inner (right side) sequence, cannot be obtained prior to the join clause itself. If this join were performed with a `join` clause, then the `Split` method would have to be called for each element. The use of multiple `from` clauses enables the query to avoid the overhead of the repeated method call. However, since `join` is optimized, in this particular case it might still be faster than using multiple `from` clauses. The results will vary depending primarily on how expensive the method call is.

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
                {student.ExamScores.Average()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

See also

LINQ query expressions
join clause
Order the results of a join clause

Handle null values in query expressions

1/5/2018 • 1 min to read • Edit Online

This example shows how to handle possible null values in source collections. An object collection such as an IEnumerable<T> can contain elements whose value is null. If a source collection is null or contains an element whose value is null, and your query does not handle null values, a NullReferenceException will be thrown when you execute the query.

Example

You can code defensively to avoid a null reference exception as shown in the following example:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

In the previous example, the `where` clause filters out all null elements in the categories sequence. This technique is independent of the null check in the join clause. The conditional expression with null in this example works because `Products.CategoryID` is of type `int?` which is shorthand for `Nullable<int>`.

Example

In a join clause, if only one of the comparison keys is a nullable value type, you can cast the other to a nullable type in the query expression. In the following example, assume that `EmployeeID` is a column that contains values of type `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

See also

Nullable<T>
LINQ query expressions
Nullable types

Handle exceptions in query expressions

1/5/2018 • 2 min to read • Edit Online

It is possible to call any method in the context of a query expression. However, we recommend that you avoid calling any method in a query expression that can create a side effect such as modifying the contents of the data source or throwing an exception. This example shows how to avoid raising exceptions when you call methods in a query expression without violating the general .NET Framework guidelines on exception handling. Those guidelines state that it is acceptable to catch a specific exception when you understand why it will be thrown in a given context. For more information, see Best Practices for Exceptions.

The final example shows how to handle those cases when you must throw an exception during execution of a query.

Example

The following example shows how to move exception handling code outside a query expression. This is only possible when the method does not depend on any variables local to the query.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Example

In some cases, the best response to an exception that is thrown from within a query might be to stop the query execution immediately. The following example shows how to handle exceptions that might be thrown from inside a query body. Assume that `SomeMethodThatMightThrow` can potentially cause an exception that requires the query execution to stop.

Note that the `try` block encloses the `foreach` loop, and not the query itself. This is because the `foreach` loop is the point at which the query is actually executed. For more information, see Introduction to LINQ queries.

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

See Also

LINQ query expressions