Visual Studio 2010 - Visual C#

**LINQ Query Expressions (C# Programming Guide)**

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language (also in Visual Basic and potentially any other .NET language). With LINQ, a query is now a first-class language construct, just like classes, methods, events and so on.

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* introduced in C# 3.0. By using query syntax, you can perform even complex 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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

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

For more information about the basics of LINQ in C#, see [Getting Started with LINQ in C#](http://msdn.microsoft.com/en-us/library/bb397933.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifQuery 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. For more information, see [Getting Started with LINQ in C#](http://msdn.microsoft.com/en-us/library/bb397933.aspx).
* 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 (C#)](http://msdn.microsoft.com/en-us/library/bb397924.aspx).
* A query is not executed until you iterate over the query variable in a **foreach** statement. For more information, see [Introduction to LINQ Queries (C#)](http://msdn.microsoft.com/en-us/library/bb397906.aspx).
* 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](http://msdn.microsoft.com/en-us/library/ms228593.aspx) and [Standard Query Operators Overview](http://msdn.microsoft.com/en-us/library/bb397896.aspx).
* 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](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count.aspx) or [Max](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.max.aspx), 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 [LINQ Query Syntax versus Method Syntax (C#)](http://msdn.microsoft.com/en-us/library/bb397947.aspx).
* Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) queries are compiled to delegates. [IQueryable](http://msdn.microsoft.com/en-us/library/system.linq.iqueryable.aspx) and [IQueryable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/bb351562.aspx) queries are compiled to expression trees. For more information, see [Expression Trees (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/bb397951.aspx).

Visual Studio 2010 - Visual C#

**Query Expression Basics (C# Programming Guide)**

Updated: May 2010

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. A SQL database table contains a sequence of rows. Similarly, an ADO.NET [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) contains a sequence of [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx) objects. 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<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) or [IQueryable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/bb351562.aspx) collection. In LINQ to XML, the source data is made visible as an **IEnumerable**<[XElement](http://msdn.microsoft.com/en-us/library/system.xml.linq.xelement.aspx)>. In LINQ to DataSet, it is an **IEnumerable**<[DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx)>. In LINQ to SQL, it is an **IEnumerable** or **IQueryable** of whatever custom objects you have defined to represent the data in the SQL table.

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[]**):

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl64_ctl00_ctl06_code');" \o "Copy Code)

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 transform from an **int** to a **string**. Note the new type of highScoresQuery.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl64_ctl00_ctl07_code');" \o "Copy Code)

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:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl64_ctl00_ctl08_code');" \o "Copy Code)

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 store the query separate from the query that stores a result.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl64_ctl00_ctl09_code');" \o "Copy Code)

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](http://msdn.microsoft.com/en-us/library/bb383978.aspx) clause and must end with a [select](http://msdn.microsoft.com/en-us/library/bb384087.aspx) or [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) clause. Between the first **from** clause and the last **select** or **group** clause, it can contain one or more of these optional clauses: [where](http://msdn.microsoft.com/en-us/library/bb311043.aspx), [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx), [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx), [let](http://msdn.microsoft.com/en-us/library/bb383976.aspx) and even additional [from](http://msdn.microsoft.com/en-us/library/bb383978.aspx) clauses. You can also use the [into](http://msdn.microsoft.com/en-us/library/bb311045.aspx) 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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl09_code');" \o "Copy Code)

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:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl10_code');" \o "Copy Code)

//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 through each is initialized with a query. They are not query variables because they store results:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl11_code');" \o "Copy Code)

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();

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();

|  |
| --- |
| **NoteNote** |
| In the LINQ documentation, variables that store a query have the word "query" as part of their names. Variables that store an actual result do not have "query" in their names. |

For more information about the different ways to express queries, see [LINQ Query Syntax versus Method Syntax (C#)](http://msdn.microsoft.com/en-us/library/bb397947.aspx).

**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](http://msdn.microsoft.com/en-us/library/bb384087.aspx). However, you can also use the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) 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:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl16_code');" \o "Copy Code)

// 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 (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384061.aspx) and [Type Relationships in LINQ Query Operations (C#)](http://msdn.microsoft.com/en-us/library/bb397924.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl19_code');" \o "Copy Code)

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:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl20_code');" \o "Copy Code)

IEnumerable<City> cityQuery =

from country in countries

from city in country.Cities

where city.Population > 10000

select city;

For more information, see [from clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb383978.aspx).

**Ending a Query Expression**

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

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl22_code');" \o "Copy Code)

var queryCountryGroups =

from country in countries

group country by country.Name[0];

For more information about grouping, see [group clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb384063.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl24_code');" \o "Copy Code)

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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl25_code');" \o "Copy Code)

// 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 (C# Reference)](http://msdn.microsoft.com/en-us/library/bb384087.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl27_code');" \o "Copy Code)

// percentileQuery is an IEnumerable<IGrouping<int, Country>>

var percentileQuery =

from country in countries

let percentile = (int) country.Population / 10000000

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 (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311045.aspx).

**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 two predicates.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl29_code');" \o "Copy Code)

IEnumerable<City> queryCityPop =

from city in cities

where city.Population < 200000 && city.Population > 100000

select city;

For more information, see [where clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311043.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl31_code');" \o "Copy Code)

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 (C# Reference)](http://msdn.microsoft.com/en-us/library/bb383982.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl33_code');" \o "Copy Code)

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](http://msdn.microsoft.com/en-us/library/bb311045.aspx) keyword. For more information, see [join clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311040.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl36_code');" \o "Copy Code)

string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };

IEnumerable<string> queryFirstNames =

from name in names

let firstName = name.Split(new char[] { ' ' })[0]

select firstName;

foreach (string s in queryFirstNames)

Console.Write(s + " ");

//Output: Svetlana Claire Sven Cesar

For more information, see [let clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb383976.aspx).

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl65_ctl00_ctl38_code');" \o "Copy Code)

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()

};

Visual Studio 2010 - Visual C#

**How to: Write LINQ Queries in C#**

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.

|  |
| --- |
| **NoteNote** |
| These queries operate on simple in-memory collections; however, the basic syntax is identical to that used in LINQ to SQL 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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl00_code');" \o "Copy Code)

// 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<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). 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. For more information, see [Introduction to LINQ Queries (C#)](http://msdn.microsoft.com/en-us/library/bb397906.aspx).

**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](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.sum.aspx), [Max](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.max.aspx), [Min](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.min.aspx), [Average](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.average.aspx), 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:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl07_code');" \o "Copy Code)

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);

If the method has parameters, these are provided in the form of a [lambda](http://msdn.microsoft.com/en-us/library/bb397687.aspx) expression, as shown in the following example:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl09_code');" \o "Copy Code)

// 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<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) 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](http://msdn.microsoft.com/en-us/library/bb383973.aspx), as shown in the following example:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl12_code');" \o "Copy Code)

// var is used for convenience in these queries

var average = numbers1.Average();

var concatenationQuery = numbers1.Concat(numbers2);

var largeNumbersQuery = numbers2.Where(c => c > 15);

**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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl13_code');" \o "Copy Code)

// 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:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl14_code');" \o "Copy Code)

var numCount = (from num in numbers...

It can be written in method syntax as follows:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl15_code');" \o "Copy Code)

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

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

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl16_code');" \o "Copy Code)

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

Visual Studio 2010 - Visual C#

**How to: Handle Exceptions in Query Expressions (C# Programming Guide)**

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 [Exception Handling](http://msdn.microsoft.com/en-us/library/ms229005.aspx).

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.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_ctl00_ctl00_code');" \o "Copy Code)

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();

}

}

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 (C#)](http://msdn.microsoft.com/en-us/library/bb397906.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_ctl00_ctl02_code');" \o "Copy Code)

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 {0}", 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.

\*/

Compiling the Code

* Create a Visual Studio project that targets the .NET Framework version 3.5. By default, the project has a reference to System.Core.dll and a **using** directive for the System.Linq namespace.
* Copy the code into your project.
* Press F5 to compile and run the program.

Press any key to exit the console window.