**Query Keywords (C# Reference)**

This section contains the contextual keywords used in query expressions.

 In This Section

|  |  |
| --- | --- |
| **Clause** | **Description** |
| [from](http://msdn.microsoft.com/en-us/library/bb383978.aspx) | Specifies a data source and a range variable (similar to an iteration variable). |
| [where](http://msdn.microsoft.com/en-us/library/bb311043.aspx) | Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators ( && or || ). |
| [select](http://msdn.microsoft.com/en-us/library/bb384087.aspx) | Specifies the type and shape that the elements in the returned sequence will have when the query is executed. |
| [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) | Groups query results according to a specified key value. |
| [into](http://msdn.microsoft.com/en-us/library/bb311045.aspx) | Provides an identifier that can serve as a reference to the results of a join, group or select clause. |
| [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) | Sorts query results in ascending or descending order based on the default comparer for the element type. |
| [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) | Joins two data sources based on an equality comparison between two specified matching criteria. |
| [let](http://msdn.microsoft.com/en-us/library/bb383976.aspx) | Introduces a range variable to store sub-expression results in a query expression. |
| [in](http://msdn.microsoft.com/en-us/library/cc713603.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [on](http://msdn.microsoft.com/en-us/library/cc713588.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [equals](http://msdn.microsoft.com/en-us/library/cc713644.aspx) | Contextual keyword in a [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) clause. |
| [by](http://msdn.microsoft.com/en-us/library/cc713645.aspx) | Contextual keyword in a [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) clause. |
| [ascending](http://msdn.microsoft.com/en-us/library/cc713606.aspx) | Contextual keyword in an [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) clause. |
| [descending](http://msdn.microsoft.com/en-us/library/cc713622.aspx) | Contextual keyword in an [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) clause. |

**from clause (C# Reference)**

A query expression must begin with a from clause. Additionally, a query expression can contain sub-queries, which also begin with a from clause. The from clause specifies the following:

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

Both the range variable and the data source are strongly typed. The data source referenced in the from clause must have a type of [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx), [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx), or a derived type such as [IQueryable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/bb351562.aspx).

In the following example, numbers is the data source and num is the range variable. Note that both variables are strongly typed even through the [var](http://msdn.microsoft.com/en-us/library/bb383973.aspx) keyword is used.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09CSharp');)

class LowNums

{

static void Main()

{

// A simple data source.

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

// Create the query.

// lowNums is an IEnumerable<int>

var lowNums = from num in numbers

where num < 5

select num;

// Execute the query.

foreach (int i in lowNums)

{

Console.Write(i + " ");

}

}

}

// Output: 4 1 3 2 0

 The Range Variable

The compiler infers the type of the range variable when the data source implements [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). For example, if the source has a type of **IEnumerable<Customer>**, then the range variable is inferred to be Customer. The only time that you must specify the type explicitly is when the source is a non-generic **IEnumerable** type such as [ArrayList](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx). For more information, see [How to: Query an ArrayList with LINQ](http://msdn.microsoft.com/en-us/library/bb397937.aspx).

In the previous example num is inferred to be of type int. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing select num, you could write select num.ToString() to cause the query expression to return a sequence of strings instead of integers. Or you could write select n + 10 to cause the expression to return the sequence 14, 11, 13, 12, 10. For more information, see [select clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb384087.aspx).

The range variable is like an iteration variable in a [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) statement except for one very important difference: a range variable never actually stores data from the source. It just a syntactic convenience that enables the query to describe what will occur when the query is executed. For more information, see [Introduction to LINQ Queries](http://msdn.microsoft.com/en-us/library/bb397906.aspx).

 Compound from Clauses

In some cases, each element in the source sequence may itself be either a sequence or contain a sequence. For example, your data source may be an **IEnumerable<Student>** where each student object in the sequence contains a list of test scores. To access the inner list within each Student element, you can use compound from clauses. The technique is like using nested [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) statements. You can add [where](http://msdn.microsoft.com/en-us/library/6b0scde8.aspx) or [orderby](http://msdn.microsoft.com/en-us/library/bb383982.aspx) clauses to either from clause to filter the results. The following example shows a sequence of Student objects, each of which contains an inner List of integers representing test scores. To access the inner list, use a compound from clause. You can insert clauses between the two from clauses if necessary.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl37CSharp');)

class CompoundFrom

{

// The element type of the data source.

public class Student

{

public string LastName { get; set; }

public List<int> Scores {get; set;}

}

static void Main()

{

// Use a collection initializer to create the data source. Note that

// each element in the list contains an inner sequence of scores.

List<Student> students = new List<Student>

{

new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},

new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},

new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},

new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},

new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}

};

// Use a compound from to access the inner sequence within each element.

// Note the similarity to a nested foreach statement.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };

// Execute the queries.

Console.WriteLine("scoreQuery:");

foreach (var student in scoreQuery)

{

Console.WriteLine("{0} Score: {1}", student.Last, student.score);

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\*

scoreQuery:

Omelchenko Score: 97

O'Donnell Score: 91

Mortensen Score: 94

Garcia Score: 97

Beebe Score: 91

\*/

 Using Multiple from Clauses to Perform Joins

A compound from clause is used to access inner collections in a single data source. However, a query can also contain multiple from clauses that generate supplemental queries from independent data sources. This technique enables you to perform certain types of join operations that are not possible by using the [join clause](http://msdn.microsoft.com/en-us/library/bb311040.aspx).

The following example shows how two from clauses can be used to form a complete cross join of two data sources.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl44CSharp');)

class CompoundFrom2

{

static void Main()

{

char[] upperCase = { 'A', 'B', 'C'};

char[] lowerCase = { 'x', 'y', 'z'};

var joinQuery1 =

from upper in upperCase

from lower in lowerCase

select new { upper, lower};

var joinQuery2 =

from lower in lowerCase

where lower != 'x'

from upper in upperCase

select new { lower, upper };

// Execute the queries.

Console.WriteLine("Cross join:");

foreach (var pair in joinQuery1)

{

Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);

}

Console.WriteLine("Filtered non-equijoin:");

foreach (var pair in joinQuery2)

{

Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

Cross join:

A is matched to x

A is matched to y

A is matched to z

B is matched to x

B is matched to y

B is matched to z

C is matched to x

C is matched to y

C is matched to z

Filtered non-equijoin:

y is matched to A

y is matched to B

y is matched to C

z is matched to A

z is matched to B

z is matched to C

\*/

**where clause (C# Reference)**

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

 Example

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08CSharp');)

class WhereSample

{

static void Main()

{

// Simple data source. Arrays support IEnumerable<T>.

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

// Simple query with one predicate in where clause.

var queryLowNums =

from num in numbers

where num < 5

select num;

// Execute the query.

foreach (var s in queryLowNums)

{

Console.Write(s.ToString() + " ");

}

}

}

//Output: 4 1 3 2 0

Within a single where clause, you can specify as many predicates as necessary by using the [&&](http://msdn.microsoft.com/en-us/library/2a723cdk.aspx) and [||](http://msdn.microsoft.com/en-us/library/6373h346.aspx) operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl11CSharp');)

class WhereSample2

{

static void Main()

{

// Data source.

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

// Create the query with two predicates in where clause.

var queryLowNums2 =

from num in numbers

where num < 5 && num % 2 == 0

select num;

// Execute the query

foreach (var s in queryLowNums2)

{

Console.Write(s.ToString() + " ");

}

}

}

// Output: 4 2 0

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl12CSharp');)

class WhereSample3

{

static void Main()

{

// Data source

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

// Create the query with a method call in the where clause.

// Note: This won't work in LINQ to SQL unless you have a

// stored procedure that is mapped to a method by this name.

var queryEvenNums =

from num in numbers

where IsEven(num)

select num;

// Execute the query.

foreach (var s in queryEvenNums)

{

Console.Write(s.ToString() + " ");

}

}

// Method may be instance method or static method.

static bool IsEven(int i)

{

return i % 2 == 0;

}

}

//Output: 4 8 6 2 0

 Remarks

The where clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A where clause may appear either before or after a [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) clause depending on whether you have to filter the source elements before or after they are grouped.

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

At compile time the where keyword is converted into a call to the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) Standard Query Operator method.

**select clause (C# Reference)**

In a query expression, the select clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the select clause itself. A query expression must terminate with either a select clause or a [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx) clause.

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl02CSharp');)

class SelectSample1

{

static void Main()

{

//Create the data source

List<int> Scores = new List<int>() { 97, 92, 81, 60 };

// Create the query.

IEnumerable<int> queryHighScores =

from score in Scores

where score > 80

select score;

// Execute the query.

foreach (int i in queryHighScores)

{

Console.Write(i + " ");

}

}

}

//Output: 97 92 81

The type of the sequence produced by the select clause determines the type of the query variable queryHighScores. In the simplest case, the select clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. For more information, see [Type Relationships in LINQ Query Operations (C#)](http://msdn.microsoft.com/en-us/library/bb397924.aspx). However, the select clause also provides a powerful mechanism for transforming (or projecting) source data into new types. For more information, see [Data Transformations with LINQ (C#)](http://msdn.microsoft.com/en-us/library/bb397914.aspx).

 Example

The following example shows all the different forms that a select clause may take. In each query, note the relationship between the select clause and the type of the query variable (studentQuery1, studentQuery2, and so on).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09CSharp');)

class SelectSample2

{

// Define some classes

public class Student

{

public string First { get; set; }

public string Last { get; set; }

public int ID { get; set; }

public List<int> Scores;

public ContactInfo GetContactInfo(SelectSample2 app, int id)

{

ContactInfo cInfo =

(from ci in app.contactList

where ci.ID == id

select ci)

.FirstOrDefault();

return cInfo;

}

public override string ToString()

{

return First + " " + Last + ":" + ID;

}

}

public class ContactInfo

{

public int ID { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public override string ToString() { return Email + "," + Phone; }

}

public class ScoreInfo

{

public double Average { get; set; }

public int ID { get; set; }

}

// The primary data source

List<Student> students = new List<Student>()

{

new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81, 60}},

new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91, 39}},

new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},

new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},

};

// Separate data source for contact info.

List<ContactInfo> contactList = new List<ContactInfo>()

{

new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},

new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="               206-555-0298        "},

new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="               206-555-1130        "},

new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="               206-555-0521        "}

};

static void Main(string[] args)

{

SelectSample2 app = new SelectSample2();

// Produce a filtered sequence of unmodified Students.

IEnumerable<Student> studentQuery1 =

from student in app.students

where student.ID > 111

select student;

Console.WriteLine("Query1: select range\_variable");

foreach (Student s in studentQuery1)

{

Console.WriteLine(s.ToString());

}

// Produce a filtered sequence of elements that contain

// only one property of each Student.

IEnumerable<String> studentQuery2 =

from student in app.students

where student.ID > 111

select student.Last;

Console.WriteLine("\r\n studentQuery2: select range\_variable.Property");

foreach (string s in studentQuery2)

{

Console.WriteLine(s);

}

// Produce a filtered sequence of objects created by

// a method call on each Student.

IEnumerable<ContactInfo> studentQuery3 =

from student in app.students

where student.ID > 111

select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range\_variable.Method");

foreach (ContactInfo ci in studentQuery3)

{

Console.WriteLine(ci.ToString());

}

// Produce a filtered sequence of ints from

// the internal array inside each Student.

IEnumerable<int> studentQuery4 =

from student in app.students

where student.ID > 111

select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range\_variable[index]");

foreach (int i in studentQuery4)

{

Console.WriteLine("First score = {0}", i);

}

// Produce a filtered sequence of doubles

// that are the result of an expression.

IEnumerable<double> studentQuery5 =

from student in app.students

where student.ID > 111

select student.Scores[0] \* 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");

foreach (double d in studentQuery5)

{

Console.WriteLine("Adjusted first score = {0}", d);

}

// Produce a filtered sequence of doubles that are

// the result of a method call.

IEnumerable<double> studentQuery6 =

from student in app.students

where student.ID > 111

select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");

foreach (double d in studentQuery6)

{

Console.WriteLine("Average = {0}", d);

}

// Produce a filtered sequence of anonymous types

// that contain only two properties from each Student.

var studentQuery7 =

from student in app.students

where student.ID > 111

select new { student.First, student.Last };

Console.WriteLine("\r\n studentQuery7: select new anonymous type");

foreach (var item in studentQuery7)

{

Console.WriteLine("{0}, {1}", item.Last, item.First);

}

// Produce a filtered sequence of named objects that contain

// a method return value and a property from each Student.

// Use named types if you need to pass the query variable

// across a method boundary.

IEnumerable<ScoreInfo> studentQuery8 =

from student in app.students

where student.ID > 111

select new ScoreInfo

{

Average = student.Scores.Average(),

ID = student.ID

};

Console.WriteLine("\r\n studentQuery8: select new named type");

foreach (ScoreInfo si in studentQuery8)

{

Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);

}

// Produce a filtered sequence of students who appear on a contact list

// and whose average is greater than 85.

IEnumerable<ContactInfo> studentQuery9 =

from student in app.students

where student.Scores.Average() > 85

join ci in app.contactList on student.ID equals ci.ID

select ci;

Console.WriteLine("\r\n studentQuery9: select result of join clause");

foreach (ContactInfo ci in studentQuery9)

{

Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);

}

// Keep the console window open in debug mode

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

Console.ReadKey();

}

}

/\* Output

Query1: select range\_variable

Claire O'Donnell:112

Sven Mortensen:113

Cesar Garcia:114

studentQuery2: select range\_variable.Property

O'Donnell

Mortensen

Garcia

studentQuery3: select range\_variable.Method

ClaireO@Contoso.com,206-555-0298

SvenMort@Contoso.com,206-555-1130

CesarGar@Contoso.com,206-555-0521

studentQuery4: select range\_variable[index]

First score = 75

First score = 88

First score = 97

studentQuery5: select expression

Adjusted first score = 82.5

Adjusted first score = 96.8

Adjusted first score = 106.7

studentQuery6: select expression2

Average = 72.25

Average = 84.5

Average = 88.25

studentQuery7: select new anonymous type

O'Donnell, Claire

Mortensen, Sven

Garcia, Cesar

studentQuery8: select new named type

ID = 112, Average = 72.25

ID = 113, Average = 84.5

ID = 114, Average = 88.25

studentQuery9: select result of join clause

ID = 114, Email = CesarGar@Contoso.com

\*/

As shown in studentQuery8 in the previous example, sometimes you might want the elements of the returned sequence to contain only a subset of the properties of the source elements. By keeping the returned sequence as small as possible you can reduce the memory requirements and increase the speed of the execution of the query. You can accomplish this by creating an anonymous type in the select clause and using an object initializer to initialize it with the appropriate properties from the source element. For an example of how to do this, see [Object and Collection Initializers (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384062.aspx).

**group clause (C# Reference)**

The group clause returns a sequence of [IGrouping<(Of <(TKey, TElement>)>)](http://msdn.microsoft.com/en-us/library/bb344977.aspx) objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type [char](http://msdn.microsoft.com/en-us/library/x9h8tsay.aspx), and is stored in the **Key** property of each [IGrouping<(Of <(TKey, TElement>)>)](http://msdn.microsoft.com/en-us/library/bb344977.aspx) object. The compiler infers the type of the key.

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08CSharp');)

// Query variable is an IEnumerable<IGrouping<char, Student>>

var studentQuery1 =

from student in students

group student by student.Last[0];

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the [into](http://msdn.microsoft.com/en-us/library/bb311045.aspx) contextual keyword. When you use into, you must continue with the query, and eventually end it with either a select statement or another group clause, as shown in the following excerpt:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl10CSharp');)

// Group students by the first letter of their last name

// Query variable is an IEnumerable<IGrouping<char, Student>>

var studentQuery2 =

from student in students

group student by student.Last[0] into g

orderby g.Key

select g;

More complete examples of the use of group with and without into are provided in the Example section of this topic.

 Enumerating the Results of a Group Query

Because the [IGrouping<(Of <(TKey, TElement>)>)](http://msdn.microsoft.com/en-us/library/bb344977.aspx) objects produced by a group query are essentially a list of lists, you must use a nested [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each item in the group itself. A group may have a key but no elements. The following is the foreach loop that executes the query in the previous code examples:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl22CSharp');)

// Iterate group items with a nested foreach. This IGrouping encapsulates

// a sequence of Student objects, and a Key of type char.

// For convenience, var can also be used in the foreach statement.

foreach (IGrouping<char, Student> studentGroup in studentQuery2)

{

Console.WriteLine(studentGroup.Key);

// Explicit type for student could also be used here.

foreach (var student in studentGroup)

{

Console.WriteLine(" {0}, {1}", student.Last, student.First);

}

}

 Key Types

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

### Grouping by string

The previous code examples used a char. A string key could easily have been specified instead, for example the complete last name:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl35CSharp');)

// Same as previous example except we use the entire last name as a key.

// Query variable is an IEnumerable<IGrouping<string, Student>>

var studentQuery3 =

from student in students

group student by student.Last;

### Grouping by bool

The following example shows the use of a bool value for a key to divide the results into two groups. Note that the value is produced by a sub-expression in the group clause.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl36CSharp');)

class GroupSample1

{

// The element type of the data source.

public class Student

{

public string First { get; set; }

public string Last { get; set; }

public int ID { get; set; }

public List<int> Scores;

}

public static List<Student> GetStudents()

{

// Use a collection initializer to create the data source. Note that each element

// in the list contains an inner sequence of scores.

List<Student> students = new List<Student>

{

new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81, 60}},

new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},

new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},

new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},

new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}

};

return students;

}

static void Main()

{

// Obtain the data source.

List<Student> students = GetStudents();

// Group by true or false.

// Query variable is an IEnumerable<IGrouping<bool, Student>>

var booleanGroupQuery =

from student in students

group student by student.Scores.Average() >= 80; //pass or fail!

// Execute the query and access items in each group

foreach (var studentGroup in booleanGroupQuery)

{

Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");

foreach (var student in studentGroup)

{

Console.WriteLine(" {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());

}

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

Low averages

Omelchenko, Svetlana:77.5

O'Donnell, Claire:72.25

Garcia, Cesar:75.5

High averages

Mortensen, Sven:93.5

Garcia, Debra:88.25

\*/

### Grouping by numeric range

The next example uses an expression to create numeric group keys that represent a percentile range. Note the use of [let](http://msdn.microsoft.com/en-us/library/bb383976.aspx) as a convenient location to store a method call result, so that you do not have to call the method two times in the group clause. Note also in the group clause that to avoid a "divide by zero" exception the code checks to make sure that the student does not have an average of zero. For more information about how to safely use methods in query expressions, see [How to: Handle Exceptions in Query Expressions (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb513730.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl39CSharp');)

class GroupSample2

{

// The element type of the data source.

public class Student

{

public string First { get; set; }

public string Last { get; set; }

public int ID { get; set; }

public List<int> Scores;

}

public static List<Student> GetStudents()

{

// Use a collection initializer to create the data source. Note that each element

// in the list contains an inner sequence of scores.

List<Student> students = new List<Student>

{

new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81, 60}},

new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},

new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},

new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},

new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}

};

return students;

}

// This method groups students into percentile ranges based on their

// grade average. The Average method returns a double, so to produce a whole

// number it is necessary to cast to int before dividing by 10.

static void Main()

{

// Obtain the data source.

List<Student> students = GetStudents();

// Write the query.

var studentQuery =

from student in students

let avg = (int)student.Scores.Average()

group student by (avg == 0 ? 0 : avg / 10) into g

orderby g.Key

select g;

// Execute the query.

foreach (var studentGroup in studentQuery)

{

int temp = studentGroup.Key \* 10;

Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);

foreach (var student in studentGroup)

{

Console.WriteLine(" {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());

}

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

Students with an average between 70 and 80

Omelchenko, Svetlana:77.5

O'Donnell, Claire:72.25

Garcia, Cesar:75.5

Students with an average between 80 and 90

Garcia, Debra:88.25

Students with an average between 90 and 100

Mortensen, Sven:93.5

\*/

### Grouping by Composite Keys

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

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl40other');)

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

Use a named type if you must pass the query variable to another method. Create a special class using auto-implemented properties for the keys, and then override the [Equals](http://msdn.microsoft.com/en-us/library/system.object.equals.aspx) and [GetHashCode](http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx) methods. You can also use a struct, in which case you do not strictly have to override those methods. For more information see [How to: Implement an Immutable Class That has Auto-Implemented Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb383979.aspx) and [How to: Query for Duplicate Files in a Directory Tree (LINQ)](http://msdn.microsoft.com/en-us/library/bb546133.aspx). The latter topic has a code example that demonstrates how to use a composite key with a named type.

 Example

The following example shows the standard pattern for ordering source data into groups when no additional query logic is applied to the groups. This is called a grouping without a continuation. The elements in an array of strings are grouped according to their first letter. The result of the query is an [IGrouping<(Of <(TKey, TElement>)>)](http://msdn.microsoft.com/en-us/library/bb344977.aspx) type that contains a public Key property of type char and an [IEnumerable<(Of <(T>)>)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) collection that contains each item in the grouping.

The result of a group clause is a sequence of sequences. Therefore, to access the individual elements within each returned group, use a nested foreach loop inside the loop that iterates the group keys, as shown in the following example.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl62CSharp');)

class GroupExample1

{

static void Main()

{

// Create a data source.

string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

// Create the query.

var wordGroups =

from w in words

group w by w[0];

// Execute the query.

foreach (var wordGroup in wordGroups)

{

Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);

foreach (var word in wordGroup)

{

Console.WriteLine(word);

}

}

// Keep the console window open in debug mode

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

Console.ReadKey();

}

}

/\* Output:

Words that start with the letter 'b':

blueberry

banana

Words that start with the letter 'c':

chimpanzee

cheese

Words that start with the letter 'a':

abacus

apple

\*/

This example shows how to perform additional logic on the groups after you have created them, by using a continuation with into. For more information, see [into (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311045.aspx). The following example queries each group to select only those whose key value is a vowel.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl64CSharp');)

class GroupClauseExample2

{

static void Main()

{

// Create the data source.

string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant", "umbrella", "anteater" };

// Create the query.

var wordGroups2 =

from w in words2

group w by w[0] into grps

where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'

|| grps.Key == 'o' || grps.Key == 'u')

select grps;

// Execute the query.

foreach (var wordGroup in wordGroups2)

{

Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);

foreach (var word in wordGroup)

{

Console.WriteLine(" {0}", word);

}

}

// Keep the console window open in debug mode

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

Console.ReadKey();

}

}

/\* Output:

Groups that start with a vowel: a

abacus

apple

anteater

Groups that start with a vowel: e

elephant

Groups that start with a vowel: u

umbrella

\*/

**into (C# Reference)**

The into contextual keyword can be used to create a temporary identifier to store the results of a [group](http://msdn.microsoft.com/en-us/library/bb384063.aspx), [join](http://msdn.microsoft.com/en-us/library/bb311040.aspx) or [select](http://msdn.microsoft.com/en-us/library/bb384087.aspx) clause into a new identifier. This identifier can itself be a generator for additional query commands. When used in a group or select clause, the use of the new identifier is sometimes referred to as a continuation.

 Example

The following example shows the use of the into keyword to enable a temporary identifier fruitGroup which has an inferred type of **IGrouping**. By using the identifier, you can invoke the [Count](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count.aspx) method on each group and select only those groups that contain two or more words.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl11CSharp');)

class IntoSample1

{

static void Main()

{

// Create a data source.

string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

// Create the query.

var wordGroups1 =

from w in words

group w by w[0] into fruitGroup

where fruitGroup.Count() >= 2

select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

// Execute the query. Note that we only iterate over the groups,

// not the items in each group

foreach (var item in wordGroups1)

{

Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);

}

// Keep the console window open in debug mode

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

Console.ReadKey();

}

}

/\* Output:

a has 2 elements.

b has 2 elements.

\*/

The use of into in a group clause is only necessary when you want to perform additional query operations on each group. For more information, see [group clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb384063.aspx).

For an example of the use of into in a join clause, see [join clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311040.aspx).

**orderby clause (C# Reference)**

In a query expression, the orderby clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending. You can also specify a custom comparer. However, it is only available by using method-based syntax. For more information, see [Sorting Data](http://msdn.microsoft.com/en-us/library/bb546145.aspx).

 Example

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl06CSharp');)

class OrderbySample1

{

static void Main()

{

// Create a delicious data source.

string[] fruits = { "cherry", "apple", "blueberry" };

// Query for ascending sort.

IEnumerable<string> sortAscendingQuery =

from fruit in fruits

orderby fruit //"ascending" is default

select fruit;

// Query for descending sort.

IEnumerable<string> sortDescendingQuery =

from w in fruits

orderby w descending

select w;

// Execute the query.

Console.WriteLine("Ascending:");

foreach (string s in sortAscendingQuery)

{

Console.WriteLine(s);

}

// Execute the query.

Console.WriteLine(Environment.NewLine + "Descending:");

foreach (string s in sortDescendingQuery)

{

Console.WriteLine(s);

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

Ascending:

apple

blueberry

cherry

Descending:

cherry

blueberry

apple

\*/

The following example performs a primary sort on the students' last names, and then a secondary sort on their first names.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07CSharp');)

class OrderbySample2

{

// The element type of the data source.

public class Student

{

public string First { get; set; }

public string Last { get; set; }

public int ID { get; set; }

}

public static List<Student> GetStudents()

{

// Use a collection initializer to create the data source. Note that each element

// in the list contains an inner sequence of scores.

List<Student> students = new List<Student>

{

new Student {First="Svetlana", Last="Omelchenko", ID=111},

new Student {First="Claire", Last="O'Donnell", ID=112},

new Student {First="Sven", Last="Mortensen", ID=113},

new Student {First="Cesar", Last="Garcia", ID=114},

new Student {First="Debra", Last="Garcia", ID=115}

};

return students;

}

static void Main(string[] args)

{

// Create the data source.

List<Student> students = GetStudents();

// Create the query.

IEnumerable<Student> sortedStudents =

from student in students

orderby student.Last ascending, student.First ascending

select student;

// Execute the query.

Console.WriteLine("sortedStudents:");

foreach (Student student in sortedStudents)

Console.WriteLine(student.Last + " " + student.First);

// Now create groups and sort the groups. The query first sorts the names

// of all students so that they will be in alphabetical order after they are

// grouped. The second orderby sorts the group keys in alpha order.

var sortedGroups =

from student in students

orderby student.Last, student.First

group student by student.Last[0] into newGroup

orderby newGroup.Key

select newGroup;

// Execute the query.

Console.WriteLine(Environment.NewLine + "sortedGroups:");

foreach (var studentGroup in sortedGroups)

{

Console.WriteLine(studentGroup.Key);

foreach (var student in studentGroup)

{

Console.WriteLine(" {0}, {1}", student.Last, student.First);

}

}

// Keep the console window open in debug mode

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

Console.ReadKey();

}

}

/\* Output:

sortedStudents:

Garcia Cesar

Garcia Debra

Mortensen Sven

O'Donnell Claire

Omelchenko Svetlana

sortedGroups:

G

Garcia, Cesar

Garcia, Debra

M

Mortensen, Sven

O

O'Donnell, Claire

Omelchenko, Svetlana

\*/

 Remarks

At compile time, the orderby clause is translated to a call to the [OrderBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.orderby.aspx) method. Multiple keys in the orderby clause translate to [ThenBy](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.thenby.aspx) method calls.

**join clause (C# Reference)**

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

A join clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The join clause compares the specified keys for equality by using the special equals keyword. All joins performed by the join clause are equijoins. The shape of the output of a join clause depends on the specific type of join you are performing. The following are three most common join types:

* Inner join
* Group join
* Left outer join

 Inner Join

The following example shows a simple inner equijoin. This query produces a flat sequence of “product name / category” pairs. The same category string will appear in multiple elements. If an element from categories has no matching products, that category will not appear in the results.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl05CSharp');)

var innerJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID

select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence

For more information, see [How to: Perform Inner Joins (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397941.aspx).

 Group Join

A join clause with an into expression is called a group join.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl12CSharp');)

var innerGroupJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

select new { CategoryName = category.Name, Products = prodGroup };

A group join produces a hierarchical result sequence, which associates elements in the left source sequence with one or more matching elements in the right side source sequence. A group join has no equivalent in relational terms; it is essentially a sequence of object arrays.

If no elements from the right source sequence are found to match an element in the left source, the join clause will produce an empty array for that item. Therefore, the group join is still basically an inner-equijoin except that the result sequence is organized into groups.

If you just select the results of a group join, you can access the items, but you cannot identify the key that they match on. Therefore, it is generally more useful to select the results of the group join into a new type that also has the key name, as shown in the previous example.

You can also, of course, use the result of a group join as the generator of another subquery:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl13CSharp');)

var innerGroupJoinQuery2 =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

from prod2 in prodGroup

where prod2.UnitPrice > 2.50M

select prod2;

For more information, see [How to: Perform Grouped Joins (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397905.aspx).

 Left Outer Join

In a left outer join, all the elements in the left source sequence are returned, even if no matching elements are in the right sequence. To perform a left outer join in LINQ, use the **DefaultIfEmpty** method in combination with a group join to specify a default right-side element to produce if a left-side element has no matches. You can use null as the default value for any reference type, or you can specify a user-defined default type. In the following example, a user-defined default type is shown:

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl19CSharp');)

var leftOuterJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

from item in prodGroup.DefaultIfEmpty(new Product{Name = String.Empty, CategoryID = 0})

select new { CatName = category.Name, ProdName = item.Name };

For more information, see [How to: Perform Left Outer Joins (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397895.aspx).

 The equals operator

A join clause performs an equijoin. In other words, you can only base matches on the equality of two keys. Other types of comparisons such as "greater than" or "not equals" are not supported. To make clear that all joins are equijoins, the join clause uses the equals keyword instead of the == operator. The equals keyword can only be used in a join clause and it differs from the == operator in one important way. With equals, the left key consumes the outer source sequence, and the right key consumes the inner source. The outer source is only in scope on the left side of equals and the inner source sequence is only in scope on the right side.

 Non-Equijoins

You can perform non-equijoins, cross joins, and other custom join operations by using multiple from clauses to introduce new sequences independently into a query. For more information, see [How to: Perform Custom Join Operations (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb882533.aspx).

 Joins on object collections vs. relational tables

In a LINQ query expression, join operations are performed on object collections. Object collections cannot be "joined" in exactly the same way as two relational tables. In LINQ, explicit join clauses are only required when two source sequences are not tied by any relationship. When working with LINQ to SQL, foreign key tables are represented in the object model as properties of the primary table. For example, in the Northwind database, the Customer table has a foreign key relationship with the Orders table. When you map the tables to the object model, the Customer class has an Orders property that contains the collection of Orders associated with that Customer. In effect, the join has already been done for you.

For more information about querying across related tables in the context of LINQ to SQL, see [How to: Map Database Relationships (LINQ to SQL)](http://msdn.microsoft.com/en-us/library/bb386950.aspx).

 Composite Keys

You can test for equality of multiple values by using a composite key. For more information, see [How to: Join by Using Composite Keys (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb907099.aspx). Composite keys can be also used in a group clause.

 Example

The following example compares the results of an inner join, a group join, and a left outer join on the same data sources by using the same matching keys. Some extra code is added to these examples to clarify the results in the console display.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl38CSharp');)

class JoinDemonstration

{

#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(string[] args)

{

JoinDemonstration app = new JoinDemonstration();

app.InnerJoin();

app.GroupJoin();

app.GroupInnerJoin();

app.GroupJoin3();

app.LeftOuterJoin();

app.LeftOuterJoin2();

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

void InnerJoin()

{

// Create the query that selects

// a property from each element.

var innerJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID

select new { Category = category.ID, Product = prod.Name };

Console.WriteLine("InnerJoin:");

// Execute the query. Access results

// with a simple foreach statement.

foreach (var item in innerJoinQuery)

{

Console.WriteLine("{0,-10}{1}", item.Product, item.Category);

}

Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());

Console.WriteLine(System.Environment.NewLine);

}

void GroupJoin()

{

// This is a demonstration query to show the output

// of a "raw" group join. A more typical group join

// is shown in the GroupInnerJoin method.

var groupJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

select prodGroup;

// Store the count of total items (for demonstration only).

int totalItems = 0;

Console.WriteLine("Simple GroupJoin:");

// A nested foreach statement is required to access group items.

foreach (var prodGrouping in groupJoinQuery)

{

Console.WriteLine("Group:");

foreach (var item in prodGrouping)

{

totalItems++;

Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);

}

}

Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems, groupJoinQuery.Count());

Console.WriteLine(System.Environment.NewLine);

}

void GroupInnerJoin()

{

var groupJoinQuery2 =

from category in categories

orderby category.ID

join prod in products on category.ID equals prod.CategoryID into prodGroup

select new

{

Category = category.Name,

Products = from prod2 in prodGroup

orderby prod2.Name

select prod2

};

//Console.WriteLine("GroupInnerJoin:");

int totalItems = 0;

Console.WriteLine("GroupInnerJoin:");

foreach (var productGroup in groupJoinQuery2)

{

Console.WriteLine(productGroup.Category);

foreach (var prodItem in productGroup.Products)

{

totalItems++;

Console.WriteLine(" {0,-10} {1}", prodItem.Name, prodItem.CategoryID);

}

}

Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems, groupJoinQuery2.Count());

Console.WriteLine(System.Environment.NewLine);

}

void GroupJoin3()

{

var groupJoinQuery3 =

from category in categories

join product in products on category.ID equals product.CategoryID into prodGroup

from prod in prodGroup

orderby prod.CategoryID

select new { Category = prod.CategoryID, ProductName = prod.Name };

//Console.WriteLine("GroupInnerJoin:");

int totalItems = 0;

Console.WriteLine("GroupJoin3:");

foreach (var item in groupJoinQuery3)

{

totalItems++;

Console.WriteLine(" {0}:{1}", item.ProductName, item.Category);

}

Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems, groupJoinQuery3.Count());

Console.WriteLine(System.Environment.NewLine);

}

void LeftOuterJoin()

{

// Create the query.

var leftOuterQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

// Store the count of total items (for demonstration only).

int totalItems = 0;

Console.WriteLine("Left Outer Join:");

// A nested foreach statement is required to access group items

foreach (var prodGrouping in leftOuterQuery)

{

Console.WriteLine("Group:", prodGrouping.Count());

foreach (var item in prodGrouping)

{

totalItems++;

Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);

}

}

Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());

Console.WriteLine(System.Environment.NewLine);

}

void LeftOuterJoin2()

{

// Create the query.

var leftOuterQuery2 =

from category in categories

join prod in products on category.ID equals prod.CategoryID into prodGroup

from item in prodGroup.DefaultIfEmpty()

select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());

// Store the count of total items

int totalItems = 0;

Console.WriteLine("Left Outer Join 2:");

// Groups have been flattened.

foreach (var item in leftOuterQuery2)

{

totalItems++;

Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);

}

Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);

/\*Output:

InnerJoin:

Cola 1

Tea 1

Mustard 2

Pickles 2

Carrots 3

Bok Choy 3

Peaches 5

Melons 5

InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:

Group:

Cola 1

Tea 1

Group:

Mustard 2

Pickles 2

Group:

Carrots 3

Bok Choy 3

Group:

Group:

Peaches 5

Melons 5

Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:

Beverages

Cola 1

Tea 1

Condiments

Mustard 2

Pickles 2

Vegetables

Bok Choy 3

Carrots 3

Grains

Fruit

Melons 5

Peaches 5

GroupInnerJoin: 8 items in 5 named groups

GroupJoin3:

Cola:1

Tea:1

Mustard:2

Pickles:2

Carrots:3

Bok Choy:3

Peaches:5

Melons:5

GroupJoin3: 8 items in 1 group

Left Outer Join:

Group:

Cola 1

Tea 1

Group:

Mustard 2

Pickles 2

Group:

Carrots 3

Bok Choy 3

Group:

Nothing! 4

Group:

Peaches 5

Melons 5

LeftOuterJoin: 9 items in 5 groups

LeftOuterJoin2: 9 items in 1 group

Left Outer Join 2:

Cola 1

Tea 1

Mustard 2

Pickles 2

Carrots 3

Bok Choy 3

Nothing! 4

Peaches 5

Melons 5

LeftOuterJoin2: 9 items in 1 group

Press any key to exit.

\*/

 Remarks

A join clause that is not followed by into is translated into a [Join](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.join.aspx) method call. A join clause that is followed by into is translated to a [GroupJoin](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.groupjoin.aspx) method call.

**let clause (C# Reference)**

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

 Example

In the following example let is used in two ways:

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl04CSharp');)

class LetSample1

{

static void Main()

{

string[] strings =

{

"A penny saved is a penny earned.",

"The early bird catches the worm.",

"The pen is mightier than the sword."

};

// Split the sentence into an array of words

// and select those whose first letter is a vowel.

var earlyBirdQuery =

from sentence in strings

let words = sentence.Split(' ')

from word in words

let w = word.ToLower()

where w[0] == 'a' || w[0] == 'e'

|| w[0] == 'i' || w[0] == 'o'

|| w[0] == 'u'

select word;

// Execute the query.

foreach (var v in earlyBirdQuery)

{

Console.WriteLine("\"{0}\" starts with a vowel", v);

}

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

"A" starts with a vowel

"is" starts with a vowel

"a" starts with a vowel

"earned." starts with a vowel

"early" starts with a vowel

"is" starts with a vowel

\*/

**in (C# Reference)**

Updated: July 2008

The in contextual keyword is used in two contexts:

* [foreach](http://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) statements
* [join clauses](http://msdn.microsoft.com/en-us/library/bb311040.aspx) in query expressions

**on (C# Reference)**

Updated: July 2008

The on contextual keyword is used in the [join clause](http://msdn.microsoft.com/en-us/library/bb311040.aspx) of a query expression to specify the join condition.

 Example

The following example shows the use of on in a join clause.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl05CSharp');)

var innerJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID

select new { ProductName = prod.Name, Category = category.Name };

**equals (C# Reference)**

Updated: July 2008

The equals contextual keyword is used in a join clause in a query expression to compare the elements of two sequences. For more information, see [join clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb311040.aspx).

 Example

The following example shows the use of the equals keyword in a join clause.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl05CSharp');)

var innerJoinQuery =

from category in categories

join prod in products on category.ID equals prod.CategoryID

select new { ProductName = prod.Name, Category = category.Name };

**by (C# Reference)**

Updated: July 2008

The by contextual keyword is used in the group clause in a query expression to specify pecify how the returned items will be grouped.. For more information, see [group clause (C# Reference)](http://msdn.microsoft.com/en-us/library/bb384063.aspx).

 Example

The following example shows the use of the by contextual keyword in a group clause to specify that the students should be grouped according to the first letter of the last name of each student.

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl05CSharp');)

var query = from student in students

group student by student.LastName[0];

**ascending (C# Reference)**

Updated: July 2008

The ascending contextual keyword is used in the [orderby clause](http://msdn.microsoft.com/en-us/library/bb383982.aspx) in query expressions to specify that the sort order is from smallest to largest. Because ascending is the default sort order, you do not have to specify it.

 Example

The following example shows the use of ascending in an [orderby clause](http://msdn.microsoft.com/en-us/library/bb383982.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07CSharp');)

IEnumerable<string> sortAscendingQuery =

from vegetable in vegetables

orderby vegetable ascending

select vegetable;

**descending (C# Reference)**

Updated: July 2008

The descending contextual keyword is used in the [orderby clause](http://msdn.microsoft.com/en-us/library/bb383982.aspx) in query expressions to specify that the sort order is from largest to smallest.

 Example

The following example shows the use of descending in an [orderby clause](http://msdn.microsoft.com/en-us/library/bb383982.aspx).

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07CSharp');)

IEnumerable<string> sortDescendingQuery =

from vegetable in vegetables

orderby vegetable descending

select vegetable;