The “SQL equivalents” column in the reference tables in this chapter do not necessarily correspond to what an `IQueryable` implementation such as LINQ to SQL will produce. Rather, it indicates what you’d typically use to do the same job if you were writing the SQL query yourself. Where there is no simple translation, the column is left blank. Where there is no translation at all, the column reads “Exception thrown”.

`Enumerable` implementation code, when shown, excludes checking for null arguments and indexing predicates.

With each of the filtering methods, you always end up with either the same number

With each of the filtering methods, you always end up with either the same number or fewer elements than you started with. You can never get more! The elements are also identical when they come out; they are not transformed in any way.

Where

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate	`TSource => bool or (TSource,int) => bool`[a]

[a] Prohibited with LINQ to SQL and Entity Framework.

Query syntax

where *bool-expression*

Enumerable Where implementation

Enumerable.Where implementation

The internal implementation of Enumerable.Where, null checking aside, is functionally equivalent to the following:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Overview

Where returns the elements from the input sequence that satisfy the given predicate.

For instance:

For instance:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));
```

```
// Result: { "Harry", "Mary", "Jay" }
```

In query syntax:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith ("y")
                            select n;
```

A where clause can appear more than once in a query and be interspersed with let clauses:

```
from n in names
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
select u;
```

```
// Result: { "HARRY", "MARY" }
```

Standard C# scoping rules apply to such queries. In other words, you cannot refer to a variable prior to declaring it with an range variable or a `let` clause.

Indexed filtering

`Where`'s predicate optionally accepts a second argument, of type `int`. This is fed with the position of each element within the input sequence, allowing the predicate to use this information in its filtering decision. For example, the following skips every the position of each element within the input sequence, allowing the predicate to use this information in its filtering decision. For example, the following skips every second element:

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 == 0);
```

```
// Result: { "Tom", "Harry", "Jay" }
```

An exception is thrown if you use indexed filtering in LINQ to SQL or EF.

SQL LIKE comparisons in LINQ to SQL and EF

The following methods on string translate to SQL's LIKE operator:

```
Contains, StartsWith, EndsWith
```

For instance, c.Name.Contains ("abc") translates to customer.Name LIKE '%abc%' (or more accurately, a parameterized version of this). Contains lets you compare only a against a locally evaluated expression; to compare against another column, you must use the SqlMethods.Like method:

against a locally evaluated expression; to compare against another column, you must use the SqlMethods.Like method:

```
... where SqlMethods.Like (c.Description, "%" + c.Name + "%")
```

SqlMethods.Like also lets you perform more complex comparisons (e.g., LIKE 'abc%def%').

< and > string comparisons in LINQ to SQL and EF

You can perform *order* comparison on strings with string's CompareTo method; this maps to SQL's < and > operators:

```
dataContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

WHERE x IN (..., ..., ...) in LINQ to SQL and EF

With LINQ to SQL and EF, you can apply the `Contains` operator to a local collection within a filter predicate. For instance:

```
string[] chosenOnes = { "Tom", "Jay" };

from c in dataContext.Customers
where chosenOnes.Contains (c.Name)
...
```

This maps to SQL's `IN` operator—in other words:

This maps to SQL's IN operator—in other words:

```
WHERE customer.Name IN ("Tom", "Jay")
```

If the local collection is an array of entities or nonscalar types, LINQ to SQL or EF may instead emit an EXISTS clause.

Take and Skip

Argument	Type
Source sequence	IEnumerable<TSource>
Number of elements to take or skip	int

Take emits the first *n* elements and discards the rest; Skip discards the first *n* elements and emits the rest. The two methods are useful together when implementing a web page allowing a user to navigate through a large set of matching records. For instance, suppose a user searches a book database for the term "mercury," and there are 100 matches. The following returns the first 20:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

The next query returns books 21 to 40:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

LINQ to SQL and EF translate Take and Skip to the ROW_NUMBER function in SQL Server 2005, or a TOP *n* subquery in earlier versions of SQL Server.

TakeWhile and SkipWhile

TakeWhile and SkipWhile

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate	`TSource => bool` or `(TSource,int) => bool`

`TakeWhile` enumerates the input sequence, emitting each item, until the given predicate is false. It then ignores the remaining elements:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100);   // { 3, 5, 2 }
```

`SkipWhile` enumerates the input sequence, ignoring each item until the given predicate is false. It then emits the remaining elements:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100);   // { 234, 4, 1 }
```

`TakeWhile` and `SkipWhile` have no translation to SQL and cause a runtime error if

TakeWhile and SkipWhile have no translation to SQL and cause a runtime error if used in a LINQ-to-db query.

Distinct

Distinct returns the input sequence, stripped of duplicates. Only the default equality comparer can be used for equality comparison. The following returns distinct letters in a string:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters);                    // HeloWrd
```

We can call LINQ methods directly on a string, because string implements IEnumerable<char>.

Projecting

IEnumerable<TSource>→IEnumerable<TResult>

Method	Description	SQL equivalents

Method	Description	SQL equivalents
Select	Transforms each input element with the given lambda expression	SELECT
SelectMany	Transforms each input element, and then flattens and concatenates the resultant subsequences	INNER JOIN, LEFT OUTER JOIN, CROSS JOIN

When querying a database, Select and SelectMany are the most versatile joining constructs; for local queries, Join and Group Join are the most *efficient* joining constructs.

Select

Argument	Type
Source sequence	IEnumerable<TSource>
Result selector	TSource => TResult or (TSource,int) => TResult[a]

Result selector	`TSource => TResult` or `(TSource,int) => TResult`[a]

[a] Prohibited with LINQ to SQL and Entity Framework.

Query syntax

Query Syntax

select *projection-expression*

Enumerable implementation

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Overview

With Select, you always get the same number of elements that you started with. Each element, however, can be transformed in any manner by the lambda function.

The following selects the names of all fonts installed on the computer (from

The following selects the names of all fonts installed on the computer (from `System.Drawing`):

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;
```

```
foreach (string name in query) Console.WriteLine (name);
```

In this example, the `select` clause converts a `FontFamily` object to its name. Here's the lambda equivalent:

```
    IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

`Select` statements are often used to project into anonymous types:

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

A projection with no transformation is sometimes used with query syntax, in order to satisfy the requirement that the query end in a `select` or `group` clause. The following selects fonts supporting strikeout:

to satisfy the requirement that the query end in a select or group clause. The following selects fonts supporting strikeout:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;
```

```
foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

In such cases, the compiler omits the projection when translating to fluent syntax.

Indexed projection

The selector expression can optionally accept an integer argument, which acts as an indexer, providing the expression with the position of each input in the input sequence. This works only with local queries:

```
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

```
IEnumerable<string> query = names
  .Select ((s,i) => i + "=" + s);

//

{ "0=Tom", "1=Dick",

... }
```

Select subqueries and object hierarchies

You can nest a subquery in a `select` clause to build an object hierarchy. The following example returns a collection describing each directory under *D:\source*, with a subcollection of files under each directory:

lowing example returns a collection describing each directory under *D:\source*, with a subcollection of files under each directory:

```
DirectoryInfo[] dirs = new DirectoryInfo (@"d:\source").GetDirectories();
```

```
var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,
```

```
    Files = from f in d.GetFiles()
            where (f.Attributes & FileAttributes.Hidden) == 0
            select new { FileName = f.Name, f.Length, }
};
```

```
foreach (var dirFiles in query)
{
  Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
```

```
{
  Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
  foreach (var file in dirFiles.Files)
    Console.WriteLine ("    " + file.FileName + "Len: " + file.Length);
}
```

The inner portion of this query can be called a *correlated subquery*. A subquery is correlated if it references an object in the outer query—in this case, it references d, the directory being enumerated.

> A subquery inside a Select allows you to map one object hierarchy to another, or map a relational object model to a hierarchical object model.

With local queries, a subquery within a Select causes double-deferred execution. In our example, the files don't get filtered or projected until the inner foreach statement enumerates.

Subqueries and joins in LINQ to SQL and EF

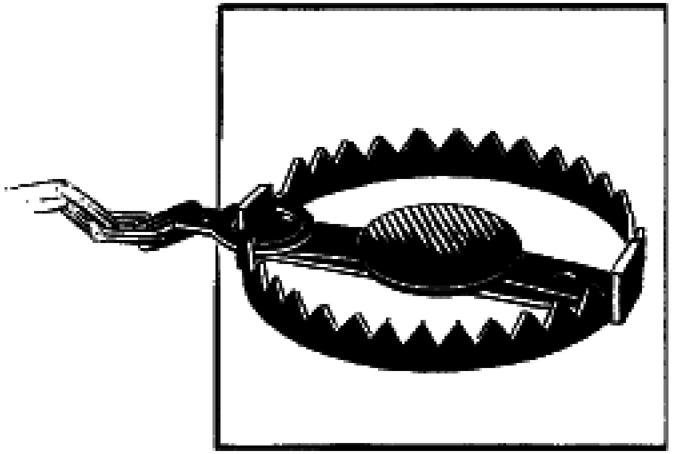
Subquery projections work well in LINQ to SQL and EF and can be used to do the

Subquery projections work well in LINQ to SQL and EF and can be used to do the work of SQL-style joins. Here's how we retrieve each customer's name along with their high-value purchases:

```
var query =
  from c in dataContext.Customers
  select new {
```

```
  from c in dataContext.Customers
  select new {
               c.Name,
               Purchases = from p in dataContext.Purchases
                           where p.CustomerID == c.ID && p.Price > 1000
                           select new { p.Description, p.Price }
             };
```

```
foreach (var namePurchases in query)
{
  Console.WriteLine ("Customer: " + namePurchases.Name);
  foreach (var purchaseDetail in namePurchases.Purchases)
    Console.WriteLine ("  - $$$: " + purchaseDetail.Price);
}
```



This style of query is ideally suited to interpreted queries. The outer query and subquery are processed as a unit, avoiding unnecessary round-tripping. With local queries, however, it's inefficient because every combination of outer and inner elements must be enumerated to get the few matching combinations. A better choice for local queries is `Join` or `GroupJoin`, described in the following sections.

This query matches up objects from two disparate collections, and it can be thought of as a "Join." The difference between this and a conventional database join (or subquery) is that we're not flattening the output into a single two-dimensional result set. We're mapping the relational data to hierarchical data, rather than to flat data.

Here's the same query simplified by using the `Purchases` association property on the `Customer` entity:

```
from c in dataContext.Customers
select new
{
  c.Name,
    {
      c.Name,
      Purchases = from p in c.Purchases    // Purchases is EntitySet<Purchase>
                  where p.Price > 1000
                  select new { p.Description, p.Price }
    };
```

Both queries are analogous to a left outer join in SQL in the sense that we get all customers in the outer enumeration, regardless of whether they have any purchases. To emulate an inner join—where customers without high-value purchases are excluded—we would need to add a filter condition on the purchases collection:

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

This is slightly untidy, however, in that we've written the same predicate (Price > 1000) twice. We can avoid this duplication with a let clause:

```
from c in dataContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

This style of query is flexible. By changing Any to Count, for instance, we can modify the query to retrieve only customers with at least two high-value purchases:

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

Projecting into concrete types

Projecting into anonymous types is useful in obtaining intermediate results, but not so useful if you want to send a result set back to a client, for instance, because so useful if you want to send a result set back to a client, for instance, because anonymous types can exist only as local variables within a method. An alternative is to use concrete types for projections, such as DataSets or custom business entity classes. A custom business entity is simply a class that you write with some properties, similar to a LINQ to SQL [Table] annotated class or an EF Entity, but designed to hide lower-level (database-related) details. You might exclude foreign key fields from business entity classes, for instance. Assuming we wrote custom entity classes called CustomerEntity and PurchaseEntity, here's how we could project into them:

```
IQueryable<CustomerEntity> query =
  from c in dataContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases =
      (from p in c.Purchases
       where p.Price > 1000
       select new PurchaseEntity {
                                  Description = p.Description,
                                  Value = p.Price
                                 }
      ).ToList()
  };

// Force query execution, converting output to a more convenient List:
```

```
// Force query execution, converting output to a more convenient List:
List<CustomerEntity> result = query.ToList();
```

Notice that so far, we've not had to use a Join or SelectMany statement. This is because we're maintaining the hierarchical shape of the data, as illustrated in Figure 9-2. With LINQ, you can often avoid the traditional SQL approach of flattening tables into a two-dimensional result set.

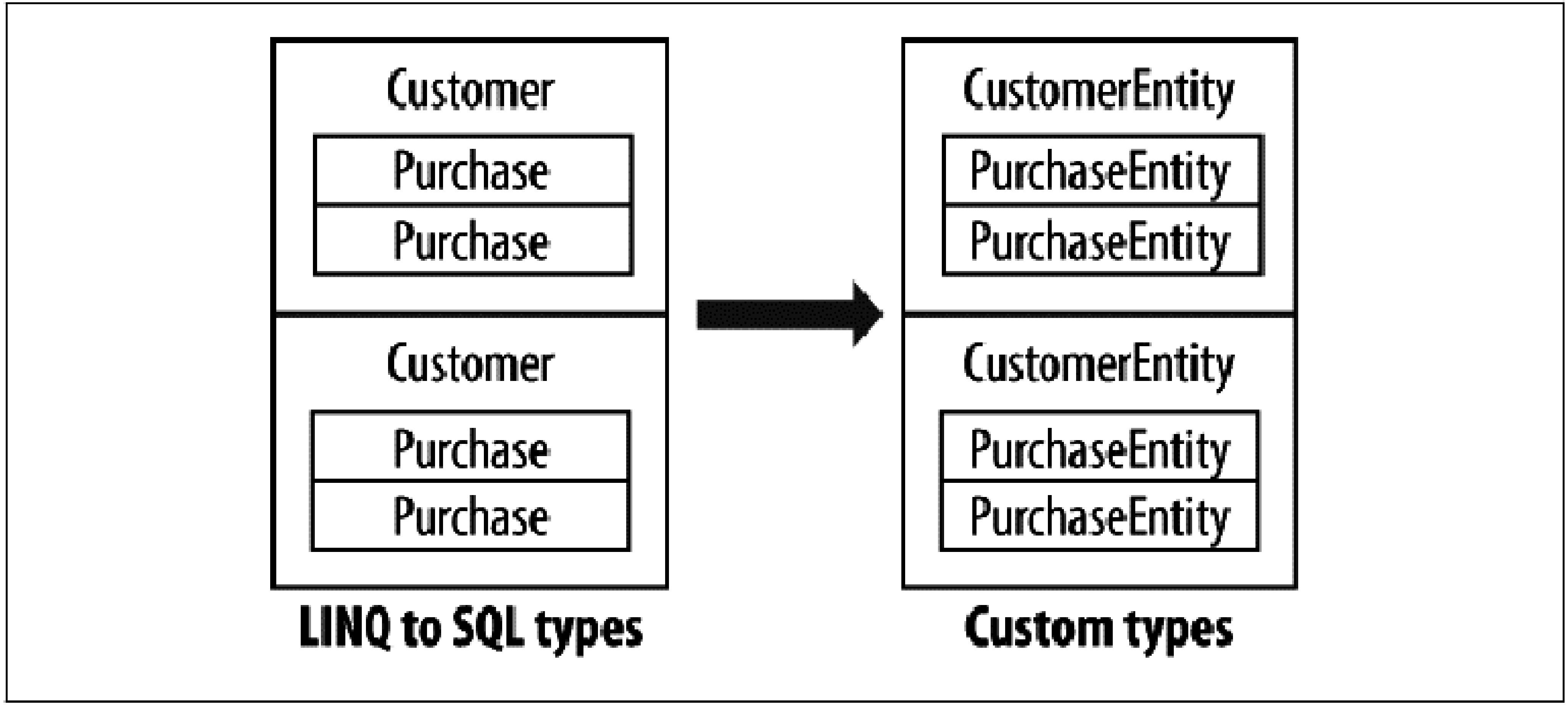


Figure 9-2. Projecting an object hierarchy

SelectMany

SelectMany

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector	`TSource => IEnumerable<TResult>` or `(TSource,int) => IEnumerable<TResult>`[a]

[a] Prohibited with LINQ to SQL

Query syntax

```
from identifier1 in enumerable-expression1
from identifier2 in enumerable-expression2
...
```

Enumerable implementation

Enumerable Implementation

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

Overview

`SelectMany` concatenates subsequences into a single flat output sequence.

Recall that for each input element, `Select` yields exactly one output element. In contrast, `SelectMany` yields *0..n* output elements. The *0..n* elements come from a subsequence or child sequence that the lambda expression must emit.

`SelectMany` can be used to expand child sequences, flatten nested collections, and join two collections into a flat output sequence. Using the conveyor belt analogy, SelectMany can be used to expand child sequences, flatten nested collections, and join two collections into a flat output sequence. Using the conveyor belt analogy, SelectMany funnels fresh material onto a conveyor belt. With SelectMany, each input

element is the *trigger* for the introduction of fresh material. The fresh material is emitted by the selector lambda expression and must be a sequence. In other words, the lambda expression must emit a *child sequence* per input *element*. The final result is a concatenation of the child sequences emitted for each input element.

Starting with a simple example, suppose we have an array of names as follows:

```
    string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

which we wish to convert to a single flat collection of words—in other words:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", Green"
```

SelectMany is ideal for this task, because we're mapping each input element to a variable number of output elements. All we must do is come up with a selector expression that converts each input element to a child sequence. string.Split does the job nicely: it takes a string and splits it into words, emitting the result as an array:

expression that converts each input element to a child sequence. string.Split does the job nicely: it takes a string and splits it into words, emitting the result as an array:

```
string testInputElement = "Anne Williams";
string[] childSequence  = testInputElement.Split();

// childSequence is { "Anne", "Williams" };
```

So, here's our SelectMany query and the result:

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());

foreach (string name in query)
  Console.Write (name + "|");  // Anne|Williams|John|Fred|Smith|Sue|Green|
```

If you replace SelectMany with Select, you get the same results in hierarchical form. The following emits a sequence of string *arrays*, requiring nested foreach statements to enumerate:

```
IEnumerable<string[]> query =
  fullNames.Select (name => name.Split());
```

```
foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write (name + "/");
```

The benefit of SelectMany is that it yields a single *flat* result sequence.

SelectMany is supported in query syntax and is invoked by having an *additional generator*—in other words, an extra from clause in the query. The from keyword has two meanings in query syntax. At the start of a query, it introduces the original range variable and input sequence. *Anywhere else* in the query, it translates to SelectMany. Here's our query in query syntax:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  from name in fullName.Split()
  select name;

// Translates to SelectMany
```

Note that the additional generator introduces a new query variable—in this case, name. The new query variable becomes the range variable from then on, and the old range variable is demoted to an *outer range variable*.

Outer range variables

In the preceding example, fullName becomes an outer range variable after SelectMany. Outer range variables remain in scope until the query either ends or reaches an into clause. The extended scope of these variables is *the* killer scenario for query syntax over fluent syntax.

To illustrate, we can take the preceding query and include fullName in the final projection:

```
IEnumerable<string> query =
  from fullName in fullNames       // fullName = outer variable
  from name in fullName.Split()    // name = range variable
  select name + " came from " + fullName;
```

```
Anne came from Anne Williams
Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

Behind the scenes, the compiler must pull some tricks to resolve outer references. A good way to appreciate this is to try writing the same query in fluent syntax. It's tricky! It gets harder still if you insert a `where` or `orderby` clause before projecting:

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
select name + " came from " + fullName;
```

The problem is that `SelectMany` emits a flat sequence of child elements—in our case, a flat collection of words. The original outer element from which it came (`fullName`) is lost. The solution is to "carry" the outer element with each child, in a a flat collection of words. The original outer element from which it came (fullName) is lost. The solution is to "carry" the outer element with each child, in a temporary anonymous type:

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

The only change here is that we're wrapping each child element (name) in an anonymous type that also contains its fullName. This is similar to how a let clause is resolved. Here's the final conversion to fluent syntax:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split()
                             .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
  .Select  (x => x.name + " came from " + x.fName);
```

Thinking in query syntax

As we just demonstrated, there are good reasons to use query syntax if you need the outer range variable. In such cases, it helps not only to use query syntax, but also to think directly in its terms.

There are two basic patterns when writing additional generators. The first is *expanding and flattening subsequences*. To do this, you call a property or method on an existing query variable in your additional generator. We did this in the previous example:

```
from fullName in fullNames
from name in fullName.Split()
```

Here, we've expanded from enumerating full names to enumerating words. An analogous LINQ-to-db query is when you expand child association properties. The following query lists all customers along with their purchases:

```
IEnumerable<string> query = from c in dataContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;
```

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Here, we've expanded each customer into a subsequence of purchases.

The second pattern is performing a *cross product* or *cross join*—where every element of one sequence is matched with every element of another. To do this, introduce a generator whose `selector` expression returns a sequence unrelated to a range variable:

variable.

```
int[] numbers = { 1, 2, 3 };
string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;

RESULT: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

This style of query is the basis of SelectMany-style *joins*.

Joining with SelectMany

You can use SelectMany to join two sequences, simply by filtering the results of a cross product. For instance, suppose we wanted to match players for a game. We could start as follows:

cross product. For instance, suppose we wanted to match players for a game. We could start as follows:

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;
```

```
RESULT: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
          "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
          "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

The query reads: “For every player, reiterate every player, selecting player 1 vs player 2”. Although we got what we asked for (a cross join), the results are not useful until we add a filter:

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo (name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;
```

```
RESULT: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

The filter predicate constitutes the *join condition*. Our query can be called a *non-equi join*, because the join condition doesn't use an equality operator.

We'll demonstrate the remaining types of joins with LINQ to SQL (they'll also work with EF except where we explicitly use a foreign key field).

SelectMany in LINQ to SQL and EF

`SelectMany` in LINQ to SQL and EF can perform cross joins, non-equi joins, inner joins, and left outer joins. You can use `SelectMany` with both predefined associations and ad hoc relationships—just as with `Select`. The difference is that `SelectMany` returns a flat rather than a hierarchical result set.

A LINQ-to-db cross join is written just as in the preceding section. The following query matches every customer to every purchase (a cross join):

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

```
              select c.Name + " might have bought a " + p.Description;
```

More typically, though, you'd want to match customers to their own purchases only. You achieve this by adding a `where` clause with a joining predicate. This results in a standard SQL-style equi-join:

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

This translates well to SQL. In the next section, we'll see how it extends to support outer joins. Reformulating such queries with LINQ's `Join` operator actually makes them *less* extensible—LINQ is opposite to SQL in this sense.

If you have association properties for relationships in your entities, you can express the same query by expanding the subcollection instead of filtering the cross product:

```
from c in dataContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

Entity Framework doesn't expose foreign keys in the entities, so for recognized relationships you *must* use its association properties rather than joining manually as we did previously.

The advantage is that we've eliminated the joining predicate. We've gone from filtering a cross product to expanding and flattening. Both queries, however, will result in the same SQL.

You can add `where` clauses to such a query for additional filtering. For instance, if we wanted only customers whose names started with "T", we could filter as follows:

```
from c in dataContext.Customers
from c in dataContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

This LINQ-to-db query would work equally well if the where clause is moved one line down. If it is a local query, however, moving the where clause down would make it less efficient. With local queries, you should filter *before* joining.

You can introduce new tables into the mix with additional from clauses. For instance, if each purchase had purchase item child rows, you could produce a flat result set of customers with their purchases, each with their purchase detail lines as follows:

```
from c in dataContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.DetailLine };
```

Each from clause introduces a new *child* table. To include data from a *parent* table (via an association property), you don't add a from clause—you simply navigate to

Each from clause introduces a new *child* table. To include data from a *parent* table (via an association property), you don't add a from clause—you simply navigate to the property. For example, if each customer has a salesperson whose name you want to query, just do this:

```
from c in dataContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

You don't use SelectMany in this case because there's no subcollection to flatten. Parent association properties return a single item.

Outer joins with SelectMany

We saw previously that a Select subquery yields a result analogous to a left outer join:

```
from c in dataContext.Customers
select new {
             c.Name,
             Purchases = from p in c.Purchases
```

```
c.Name,
Purchases = from p in c.Purchases
            where p.Price > 1000
```

```
};
```

```
};
```

```
select new { p.Description, p.Price }
```

In this example, every outer element (customer) is included, regardless of whether the customer has any purchases. But suppose we rewrite this query with `SelectMany`, so we can obtain a single flat collection rather than a hierarchical result set:

```
from c in dataContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

In the process of flattening the query, we've switched to an inner join: customers are now included only for whom one or more high-value purchases exist. To get a left outer join with a flat result set, we must apply the `DefaultIfEmpty` query operator on the inner sequence. This method returns null if its input sequence has no elements. Here's such a query, price predicate aside:

ments. Here's such a query, price predicate aside:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

This works perfectly with LINQ to SQL and EF, returning all customers, even if they have no purchases. But if we were to run this as a local query, it would crash, because when p is null, p.Description and p.Price throw a NullReferenceException. We can make our query robust in either scenario as follows:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Let's now reintroduce the price filter. We cannot use a where clause as we did before, because it would execute *after* DefaultIfEmpty:

Let's now reintroduce the price filter. We cannot use a where clause as we did before, because it would execute *after* DefaultIfEmpty:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

The correct solution is to splice the Where clause *before* DefaultIfEmpty with a subquery:

```
from c in dataContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
                c.Name,
                Descript = p == null ? null : p.Description,
                Price = p == null ? (decimal?) null : p.Price
              };
```

LINQ to SQL and EF translate this to a left outer join. This is an effective pattern for writing such queries.

LINQ to SQL and EF translate this to a left outer join. This is an effective pattern for writing such queries.

If you’re used to writing outer joins in SQL, you might be tempted to overlook the simpler option of a Select subquery for this style of query, in favor of the awkward but familiar SQL-centric flat approach. The hierarchical result set from a Select subquery is often better suited to outer join-style queries because there are no additional nulls to deal with.

Joining

IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>

Method	Description	SQL equivalents
Join	Applies a lookup strategy to match elements from two collections, emitting a flat result set	INNER JOIN

Join	Applies a lookup strategy to match elements from two collections, emitting a flat result set	INNER JOIN
GroupJoin	As above, but emits a *hierarchical* result set	INNER JOIN, LEFT OUTER JOIN

Join and GroupJoin

Join arguments

Argument	Type
Outer sequence	IEnumerable<TOuter>

Outer sequence	`IEnumerable<TOuter>`
Inner sequence	`IEnumerable<TInner>`
Outer key selector	`TOuter => TKey`
Inner key selector	`TInner => TKey`
Result selector	`(TOuter,TInner) => TResult`

GroupJoin arguments

Argument	Type
Outer sequence	`IEnumerable<TOuter>`
Inner sequence	`IEnumerable<TInner>`
Outer key selector	`TOuter => TKey`
Inner key selector	`TInner => TKey`

Inner key selector	`TInner => TKey`
Result selector	`(TOuter,`**`IEnumerable<TInner>`**`) => TResult`

Query syntax

```
from outer-var in outer-enumerable
join inner-var in inner-enumerable on outer-key-expr equals inner-key-expr
 [ into identifier ]
```

Overview

`Join` and `GroupJoin` mesh two input sequences into a single output sequence. `Join` emits flat output; `GroupJoin` emits hierarchical output.

`Join` and `GroupJoin` provide an alternative strategy to `Select` and `SelectMany`. The advantage of `Join` and `GroupJoin` is that they execute efficiently over local in-memory collections, since they first load the inner sequence into a keyed lookup, avoiding the need to repeatedly enumerate over every inner element. The disadvantage is that they offer the equivalent of inner and left outer joins only; cross joins and non-equi joins must still be done with `Select`/`SelectMany`. With LINQ to SQL and Entity Framework queries, `Join` and `GroupJoin` offer no real benefits over `Select` and `SelectMany`.

Table 9-1 summarizes the differences between each of the joining strategies.

Table 9-1 summarizes the differences between each of the joining strategies.

Table 9-1. Joining strategies

Strategy	Result shape	Local query efficiency	Inner joins	Left outer joins	Cross joins	Non-equi joins
Select + SelectMany	Flat	Bad	Yes	Yes	Yes	Yes
Select + Select	Nested	Bad	Yes	Yes	Yes	Yes

Join

Flat

Good

Yes

-

Yes

Yes

-

-

-

-

GroupJoin

Nested

Good

Yes

GroupJoin + SelectMany	Flat	Good	Yes	Yes	-	-

Join

The Join operator performs an inner join, emitting a flat output sequence.

The Join operator performs an inner join, emitting a flat output sequence.

Entity Framework hides foreign key fields, so you can't manually join across natural relationships (instead, you can query across association properties, as we described in the previous two sections).

The simplest way to demonstrate Join is with LINQ to SQL. The following query lists all customers alongside their purchases, without using an association property:

```
IQueryable<string> query =
  from c in dataContext.Customers
  join p in dataContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

The results match what we would get from a SelectMany-style query:

```
Tom bought a Bike
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

To see the benefit of Join over SelectMany, we must convert this to a local query. We can demonstrate this by first copying all customers and purchases to arrays, and then querying the arrays:

```
Customer[] customers = dataContext.Customers.ToArray();
Purchase[] purchases = dataContext.Purchases.ToArray();
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;
```

```
var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

```
                 select c.Name + " bought a " + p.Description;
```

Although both queries yield the same results, the `Join` query is considerably faster because its implementation in `Enumerable` preloads the inner collection (`purchases`) into a keyed lookup.

The query syntax for `join` can be written in general terms as follows:

```
join inner-var in inner-sequence on outer-key-expr equals inner-key-expr
```

Join operators in LINQ differentiate between the *outer sequence* and *inner sequence*. Syntactically:

-
-

The *outer sequence* is the input sequence (in this case, `customers`).

The *inner sequence* is the new collection you introduce (in this case, `purchases`).

`Join` performs inner joins, meaning customers without purchases are excluded from the output. With inner joins, you can swap the inner and outer sequences in the query and still get the same results:

the output. With inner joins, you can swap the inner and outer sequences in the query and still get the same results:

```
from p in purchases                              // p is now outer
join c in customers on p.CustomerID equals c.ID   // c is now inner
...
```

You can add further join clauses to the same query. If each purchase, for instance, has one or more purchase items, you could join the purchase items as follows:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
join pi in purchaseItems on p.ID equals pi.PurchaseID
...
```

```
// first join
// second join
```

purchases acts as the *inner* sequence in the first join and as the *outer* sequence in the second join. You could obtain the same results (inefficiently) using nested foreach statements as follows:

second join. You could obtain the same results (inefficiently) using nested foreach statements as follows:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

In query syntax, variables from earlier joins remain in scope—just as outer range variables do with SelectMany-style queries. You're also permitted to insert where and let clauses in between join clauses.

Joining on multiple keys

You can join on multiple keys with anonymous types as follows:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

For this to work, the two anonymous types must be structured identically. The compiler then implements each with the same internal type, making the joining keys compatible.

compiler then implements each with the same internal type, making the joining keys compatible.

Joining in fluent syntax

The following query syntax join:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

in fluent syntax is as follows:

```
    customers.Join (                    // outer collection
          purchases,                    // inner collection
          c => c.ID,                    // outer key selector
          p => p.CustomerID,            // inner key selector
          (c, p) => new
             { c.Name, p.Description, p.Price }    // result selector
    );
```

```
                { c.Name, p.Description, p.Price }    // result selector
    );
```

The result selector expression at the end creates each element in the output sequence. If you have additional clauses prior to projecting, such as **orderby** in this example:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

you must manufacture a temporary anonymous type in the result selector in fluent syntax. This keeps both c and p in scope following the join:

```
customers.Join (                        // outer collection
      purchases,                        // inner collection
      c => c.ID,                        // outer key selector
      p => p.CustomerID,                // inner key selector
      (c, p) => new { c, p } )          // result selector
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a " + x.p.Description);
```

Query syntax is usually preferable when joining; it’s less fiddly.

GroupJoin

GroupJoin does the same work as Join, but instead of yielding a flat result, it yields a hierarchical result, grouped by each outer element. It also allows left outer joins.

The query syntax for GroupJoin is the same as for Join, but is followed by the into keyword.

Here’s the most basic example:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
```

```
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select custPurchases;   // custPurchases is a sequence
```

An `into` clause translates to `GroupJoin` only when it appears directly after a `join` clause. After a `select` or `group` clause, it means *query continuation*. The two uses of the `into` keyword are quite different, although they have one feature in common: they both introduce a new query variable.

The result is a sequence of sequences, which we could enumerate as follows:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

This isn't very useful, however, because `outerSeq` has no reference to the outer customer. More commonly, you'd reference the outer range variable in the projection:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

This gives the same results as the following (inefficient) Select subquery:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

By default, GroupJoin does the equivalent of a left outer join. To get an inner join—where customers without purchases are excluded—filter on custPurchases:

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any()
select ...
```

```
where custPurchases.Any()
select ...
```

Clauses after a group-join `into` operate on *subsequences* of inner child elements, not *individual* child elements. This means that to filter individual purchases, you'd have to call `Where` *before* joining:

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

You can construct lambda queries with GroupJoin as you would with Join.

Flat outer joins

You run into a dilemma if you want both an outer join and a flat result set. GroupJoin gives you the outer join; Join gives you the flat result set. The solution is to first call GroupJoin, and then DefaultIfEmpty on each child sequence, and then finally SelectMany on the result:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
```

```
from cp in c.Purchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

DefaultIfEmpty emits a null value if a subsequence of purchases is empty. The second from clause translates to SelectMany. In this role, it *expands and flattens* all the purchase subsequences, concatenating them into a single sequence of purchase *elements*.

Joining with lookups

The Join and GroupJoin methods in Enumerable work in two steps. First, they load the inner sequence into a *lookup*. Second, they query the outer sequence in combination with the lookup.

A *lookup* is a sequence of groupings that can be accessed directly by key. Another way to think of it is as a dictionary of sequences—a dictionary that can accept many elements under each key. Lookups are read-only and defined by the following interface:

elements under each key. Lookups are read-only and defined by the following interface:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

The joining operators—like other sequence-emitting operators—honor deferred or lazy execution semantics. This means the lookup is not built until you begin enumerating the output sequence.

You can create and query lookups manually as an alternative strategy to using the

You can create and query lookups manually as an alternative strategy to using the joining operators, when dealing with local collections. There are a couple of benefits in doing so:

●

●

You can reuse the same lookup over multiple queries—as well as in ordinary imperative code.

Querying a lookup is an excellent way of understanding how `Join` and `GroupJoin` work.

The `ToLookup` extension method creates a lookup. The following loads all purchases into a lookup—keyed by their `CustomerID`:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p => p);
```

The first argument selects the key; the second argument selects the objects that are to be loaded as values into the lookup.

to be loaded as values into the lookup.

Reading a lookup is rather like reading a dictionary, except that the indexer returns a *sequence* of matching items, rather than a *single* matching item. The following enumerates all purchases made by the customer whose ID is 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

With a lookup in place, you can write `SelectMany`/`Select` queries that execute as efficiently as `Join`/`GroupJoin` queries. `Join` is equivalent to using `SelectMany` on a lookup:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };
```

```
Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Bike 600
Dick Phone 300
...
```

Adding a call to DefaultIfEmpty makes this into an outer join:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
 select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
            };
```

GroupJoin is equivalent to reading the lookup inside a projection:

```
from c in customers
select new {
             CustName = c.Name,
             CustPurchases = purchLookup [c.ID]
           };
```

Enumerable implementations

Enumerable implementations

Here’s the simplest valid implementation of Enumerable.Join, null checking aside:

```
public static IEnumerable <TResult> Join
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,TInner,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
    select resultSelector (outerItem, innerItem);
}
```

GroupJoin’s implementation is like that of Join, but simpler:

```
public static IEnumerable <TResult> GroupJoin
public static IEnumerable <TResult> GroupJoin
                                     <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,IEnumerable<TInner>,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    select resultSelector
     (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

Ordering

IEnumerable<TSource>→IOrderedEnumerable<TSource>

Method	Description	SQL equivalents
OrderBy, ThenBy	Sorts a sequence in ascending order	ORDER BY

Method	Description	SQL equivalents
`OrderBy, ThenBy`	Sorts a sequence in ascending order	`ORDER BY` ...
`OrderByDescending, ThenByDescending`	Sorts a sequence in descending order	`ORDER BY` ... `DESC`
`Reverse`	Returns a sequence in reverse order	Exception thrown

Ordering operators return the same elements in a different order.

OrderBy, OrderByDescending, ThenBy, and ThenByDescending

OrderBy and OrderByDescending arguments

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`

Return type = `IOrderedEnumerable<TSource>`

Return type = `IOrderedEnumerable<TSource>`

ThenBy and ThenByDescending arguments

Argument	Type
Input sequence	`IOrderedEnumerable<TSource>`
Key selector	`TSource => TKey`

Query syntax

```
orderby expression1 [descending] [, expression2 [descending] ... ]
```

Overview

`OrderBy` returns a sorted version of the input sequence, using the `keySelector` expression to make comparisons. The following query emits a sequence of names in alphabetical order:

alphabetical order:

```
IEnumerable<string> query = names.OrderBy (s => s);
```

The following sorts names by length:

```
IEnumerable<string> query = names.OrderBy (s => s.Length);

// Result: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

The relative order of elements with the same sorting key (in this case, Jay/Tom and Mary/Dick) is indeterminate—unless you append a ThenBy operator:

```
IEnumerable<string> query = names.OrderBy (s => s.Length).ThenBy (s => s);

// Result: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

ThenBy reorders only elements that had the same sorting key in the preceding sort. You can chain any number of ThenBy operators. The following sorts first by length, then by the second character, and finally by the first character:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

The equivalent in query syntax is this:

The equivalent in query syntax is this:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

LINQ also provides `OrderByDescending` and `ThenByDescending` operators, which do the same things, emitting the results in reverse order. The following LINQ-to-db query retrieves purchases in descending order of price, with those of the same price listed alphabetically:

```
dataContext.Purchases.OrderByDescending (p => p.Price)
                        .ThenBy (p => p.Description);
```

In query syntax:

```
from p in dataContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Comparers and collations

In a local query, the key selector objects themselves determine the ordering algorithm

Comparers and collations

In a local query, the key selector objects themselves determine the ordering algorithm via their default IComparable implementation (see Chapter 7). You can override the sorting algorithm by passing in an IComparer object. The following performs a case-insensitive sort:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Passing in a comparer is not supported in query syntax, nor in any way by LINQ to SQL or EF. When querying a database, the comparison algorithm is determined by the participating column's collation. If the collation is case-sensitive, you can request a case-insensitive sort by calling ToUpper in the key selector:

```
from p in dataContext.Purchases
orderby p.Description.ToUpper()
select p;
```

IOrderedEnumerable and IOrderedQueryable

IOrderedEnumerable and IOrderedQueryable

The ordering operators return special subtypes of IEnumerable<T>. Those in Enumerable return IOrderedEnumerable; those in Queryable return IOrderedQueryable. These subtypes allow a subsequent ThenBy operator to refine rather than replace the existing ordering.

The additional members that these subtypes define are not publicly exposed, so they present like ordinary sequences. The fact that they are different types comes into play when building queries progressively:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

If we instead declare query1 of type IEnumerable<string>, the second line would not compile—ThenBy requires an input of type IOrderedEnumerable<string>. You can avoid worrying about this by implicitly typing query variables:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Implicit typing can create problems of its own, though. The following will not compile:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);
```

```
// Compile-time error
```

The compiler infers query to be of type IOrderedEnumerable<string>, based on OrderBy's output sequence type. However, the Where on the next line returns an ordinary IEnumerable<string>, which cannot be assigned back to query. You can work around this either with explicit typing or by calling AsEnumerable() after OrderBy:

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);                          // OK
```

The equivalent in interpreted queries is to call AsQueryable.

Grouping

Grouping

```
IEnumerable<TSource>→IEnumerable<IGrouping<TSource,TElement>>
```

Method	Description	SQL equivalents
`GroupBy`	Groups a sequence into subsequences	`GROUP BY`

GroupBy

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`
Element selector (optional)	`TSource => TElement`
Comparer (optional)	`IEqualityComparer<TKey>`

Comparer (optional)	`IEqualityComparer<TKey>`

Query syntax

```
group element-expression by key-expression
```

Overview

`GroupBy` organizes a flat input sequence into sequences of *groups*. For example, the following organizes all the files in *c:\temp* by extension:

```
string[] files = Directory.GetFiles ("c:\\temp");
```

```
IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

Or if you're comfortable with implicit typing:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

Here's how to enumerate the result:

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine ("Extension: " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine ("   " + filename);
  foreach (string filename in grouping)
    Console.WriteLine ("   - " + filename);
}
```

```
Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Enumerable.GroupBy works by reading the input elements into a temporary dictionary of lists so that all elements with the same key end up in the same sublist. It then emits a sequence of *groupings*. A grouping is a sequence with a Key property:

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; }     // Key applies to the subsequence as a whole
```

```
{
  TKey Key { get; }    // Key applies to the subsequence as a whole
}
```

By default, the elements in each grouping are untransformed input elements, unless you specify an elementSelector argument. The following projects each input element to uppercase:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

An elementSelector is independent of the keySelector. In our case, this means that the Key on each grouping is still in its original case:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Note that the subcollections are not emitted in alphabetical order of key. GroupBy groups only; it does not *sort*; in fact, it preserves the original ordering. To sort, you must add an OrderBy operator:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
     .OrderBy (grouping => grouping.Key);
```

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
     .OrderBy (grouping => grouping.Key);
```

GroupBy has a simple and direct translation in query syntax:

```
group element-expr by key-expr
```

Here's our example in query syntax:

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

As with select, group "ends" a query—unless you add a query continuation clause:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
orderby grouping.Key
select grouping;
```

Query continuations are often useful in a **group by** query. The next query filters out groups that have fewer than five files in them:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count() < 5
select grouping;
```

A **where** after a **group by** is equivalent to **HAVING** in SQL. It applies to each subsequence or grouping as a whole, rather than the individual elements.

Sometimes you're interested purely in the result of an aggregation on a grouping and so can abandon the subsequences:

```
    string[] votes = { "Bush", "Gore", "Gore", "Bush", "Bush" };
```

```
IEnumerable<string> query = from vote in votes
                            group vote by vote into g
                            orderby g.Count() descending
                            select g.Key;
```

```
string winner = query.First();
```

```
// Bush
```

GroupBy in LINQ to SQL and EF

Grouping works in the same way when querying a database. If you have association properties set up, you'll find, however, that the need to group arises less frequently than with standard SQL. For instance, to select customers with at least two purchases, you don't need to group; the following query does the job nicely:

```
from c in dataContext.Customers
from c in dataContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

An example of when you might use grouping is to list total sales by year:

```
from p in dataContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

LINQ's grouping operators expose a superset of SQL's "GROUP BY" functionality.

Another departure from traditional SQL comes in there being no obligation to project the variables or expressions used in grouping or sorting.

Grouping by multiple keys

You can group by a composite key, using an anonymous type:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

Custom equality comparers

You can pass a custom equality comparer into `GroupBy`, in a local query, to change the algorithm for key comparison. Rarely is this required, though, because changing the key selector expression is usually sufficient. For instance, the following creates a case-insensitive grouping:

```
group name by name.ToUpper()
```

Set Operators

`IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>`

Method	Description	SQL equivalents

Method	Description	SQL equivalents
Concat	Returns a concatenation of elements in each of the two sequences	UNION ALL

Union
Intersect
Except

Returns a concatenation of elements in each of the two sequences, excluding duplicates

UNION

Returns elements present in both sequences

Returns elements present in the first, but not the second sequence

WHERE ... IN (...)

```
WHERE ... IN (...)
EXCEPT
```

or

```
WHERE ... NOT IN (...)
```

Concat and Union

`Contact` returns all the elements of the first sequence, followed by all the elements of the second. `Union` does the same, but removes any duplicates:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
```

```
IEnumerable<int>
  concat = seq1.Concat (seq2),
  union  = seq1.Union  (seq2);
```