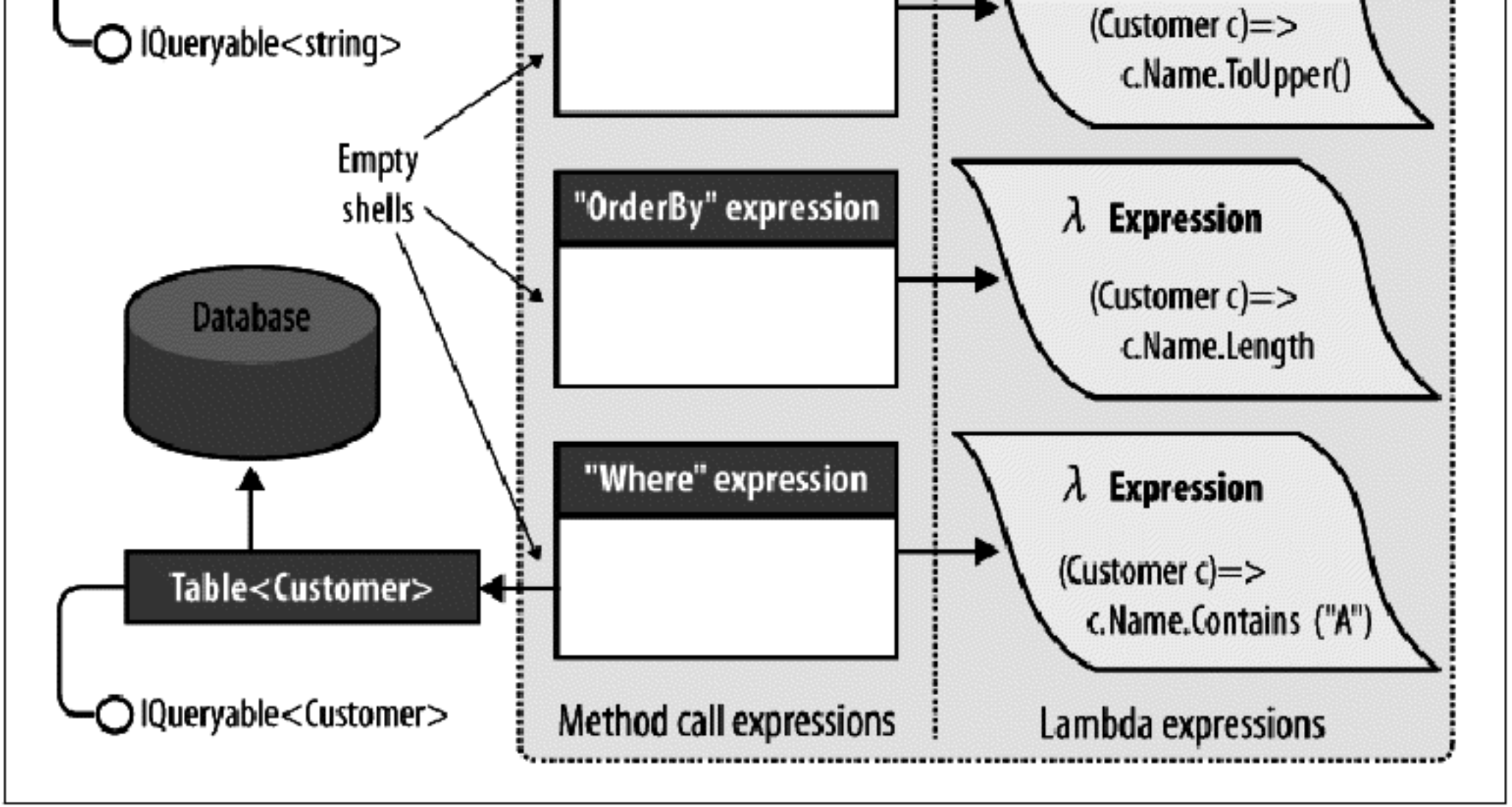


Figure 8-9. Interpreted query composition

Execution

Interpreted queries follow a deferred execution model—just like local queries. This means that the SQL statement is not generated until you start enumerating the query.

Interpreted queries follow a deferred execution model—just like local queries. This means that the SQL statement is not generated until you start enumerating the query. Further, enumerating the same query twice results in the database being queried twice.

Under the covers, interpreted queries differ from local queries in how they execute. When you enumerate over an interpreted query, the outermost sequence runs a program that traverses the entire expression tree, processing it as a unit. In our example, LINQ to SQL translates the expression tree to a SQL statement, which it then executes, yielding the results as a sequence.

To work, LINQ to SQL needs some clues as to the schema of the database. The `Table` and `Column` attributes that we applied to the `Customer` class serve just this function. The section "LINQ to SQL and Entity Framework" on page 346 describes these attributes in more detail. Entity Framework is similar except that it also requires an Entity Data Model (EDM)—an XML file describing the mapping between database and entities.

describing the mapping between database and entities.

We said previously that a LINQ query is like a production line. When you enumerate an `IQueryable` conveyor belt, though, it doesn't start up the whole production line, like with a local query. Instead, just the `IQueryable` belt starts up, with a special enumerator that calls upon a production manager. The manager reviews the entire production line—which consists not of compiled code, but of *dummies* (method call expressions) with instructions pasted to their *foreheads* (expression trees). The manager then traverses all the expressions, in this case transcribing them to a single piece of paper (a SQL statement), which it then executes, feeding the results back to the consumer. Only one belt turns; the rest of the production line is a network of empty shells, existing just to describe what has to be done.

This has some practical implications. For instance, with local queries, you can write your own query methods (fairly easily, with iterators) and then use them to supplement the predefined set. With remote queries, this is difficult, and even undesirable. If you wrote a `MyWhere` extension method accepting `IQueryable<T>`, it would be like putting your own dummy into the production line. The production manager wouldn't know what to do with your dummy. Even if you intervened at this stage, your solution would be hard-wired to a particular provider, such as LINQ to SQL, and would not work with other `IQueryable` implementations. Part of the benefit of having a standard set of methods in `Queryable` is that they define a *standard vocabulary* for querying *any* remote collection. As soon as you try to extend the vocabulary,

having a standard set of methods in `Queryable` is that they define a *standard vocabulary* for querying *any* remote collection. As soon as you try to extend the vocabulary, you're no longer interoperable.

Another consequence of this model is that an `IQueryable` provider may be unable to cope with some queries—even if you stick to the standard methods. LINQ to SQL and EF are both limited by the capabilities of the database server; some LINQ queries have no SQL translation. If you're familiar with SQL, you'll have a good intuition for what these are, although at times you have to experiment to see what causes a runtime error; it can be surprising what *does* work!

Combining Interpreted and Local Queries

A query can include both interpreted and local operators. A typical pattern is to have the local operators on the *outside* and the interpreted components on the *inside*; in other words, the interpreted queries feed the local queries. This pattern works well with LINQ-to-database queries.

For instance, suppose we write a custom extension method to pair up strings in a collection:

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
  string firstHalf = null;
  foreach (string element in source)
    if (firstHalf == null)
      firstHalf = element;
```

```
    if (firstHalf == null)
      firstHalf = element;
    else
    {
      yield return firstHalf + ", " + element;
      firstHalf = null;
    }
}
```

We can use this extension method in a query that mixes LINQ to SQL and local operators:

```
DataContext dataContext = new DataContext ("connection string");
Table<Customer> customers = dataContext.GetTable <Customer>();
```

```
IEnumerable<string> q = customers
  .Select (c => c.Name.ToUpper())
  .OrderBy (n => n)
  .Pair()                              // Local from this point on.
  .Select ((n, i) => "Pair " + i.ToString() + " = " + n);
```

```
foreach (string element in q) Console.WriteLine (element);
```

```
foreach (string element in q) Console.WriteLine (element);
```

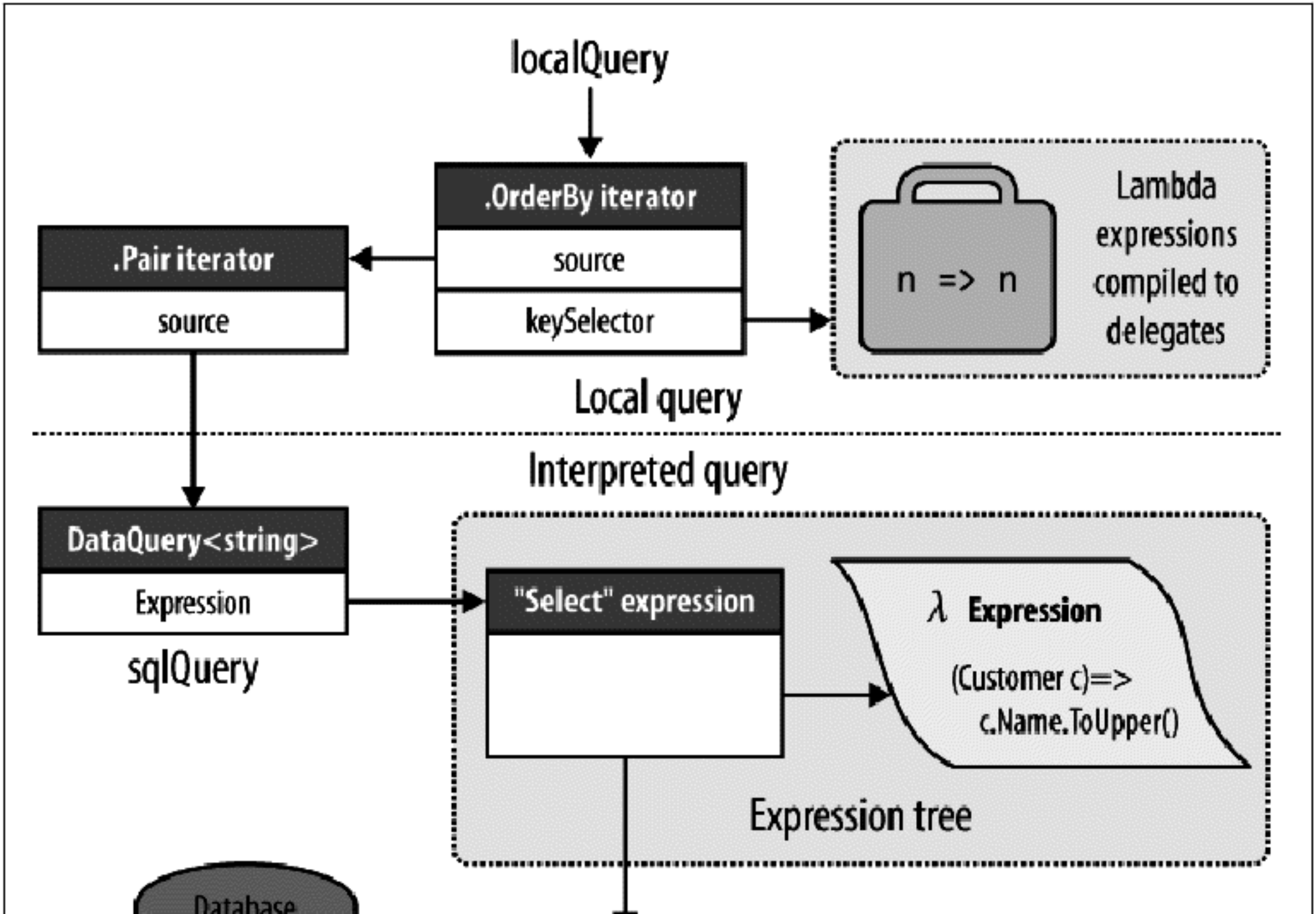
```
Pair 0 = HARRY, MARY
Pair 1 = TOM, DICK
```

Because `customers` is of a type implementing `IQueryable<T>`, the `Select` operator resolves to `Queryable.Select`. This returns an output sequence also of type `IQueryable<T>`. But the next query operator, `Pair`, has no overload accepting `IQueryable<T>`, only the less specific `IEnumerable<>`. So, it resolves to our local `Pair` method—wrapping the interpreted query in a local query. `Pair` also emits `IEnumerable`, so `OrderBy` wraps another local operator.

On the LINQ to SQL side, the resulting SQL statement is equivalent to:

```
SELECT UPPER (Name) FROM Customer ORDER BY UPPER (Name)
```

The remaining work is done locally. Figure 8-10 shows the query diagrammatically. In effect, we have a local query (on the outside), whose source is an interpreted query (the inside).


localQuery
.OrderBy iterator
source
keySelector
.Pair iterator
source
n => n
Lambda expressions compiled to delegates
Local query
Interpreted query
DataQuery<string>
Expression
sqlQuery
"Select" expression
λ Expression
(Customer c)=>
c.Name.ToUpper()
Expression tree
Database

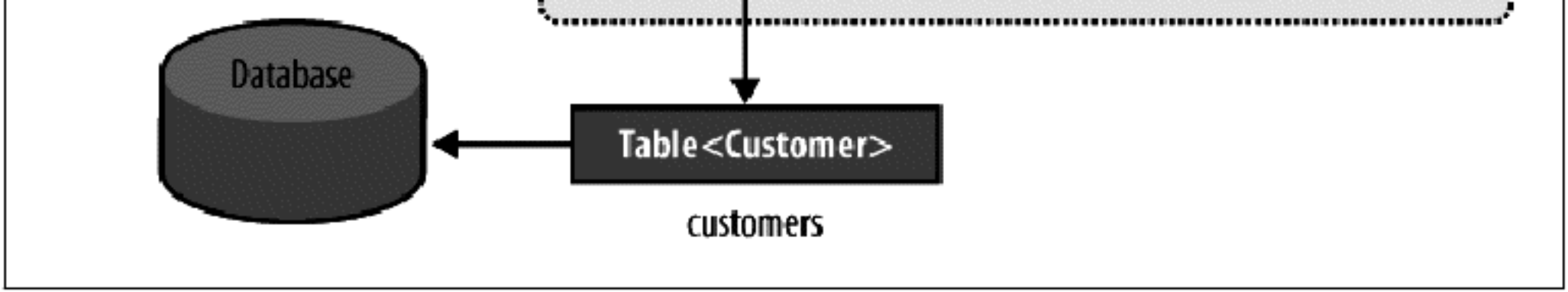


Figure 8-10. Combining local and interpreted queries

AsEnumerable

`Enumerable.AsEnumerable` is the simplest of all query operators. Here's its complete definition:

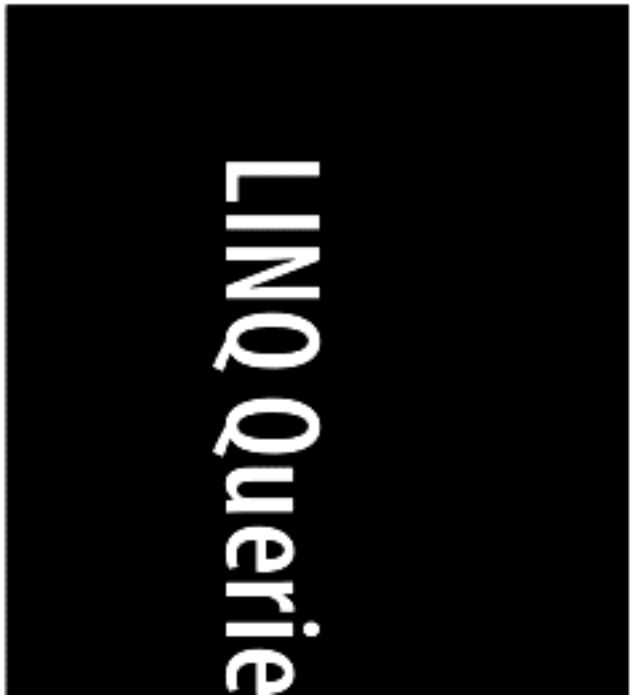
```
public static IEnumerable<TSource> AsEnumerable<TSource>
              (this IEnumerable<TSource> source)
{
    return source;
}
```

Its purpose is to cast an `IQueryable<T>` sequence to `IEnumerable<T>`, forcing subsequent query operators to bind to `Enumerable` operators instead of `Queryable` opera-

Its purpose is to cast an `IQueryable<T>` sequence to `IEnumerable<T>`, forcing subsequent query operators to bind to `Enumerable` operators instead of `Queryable` operators. This causes the remainder of the query to execute locally.

To illustrate, suppose we had a `MedicalArticles` table in SQL Server and wanted to use LINQ to SQL or EF to retrieve all articles on influenza whose abstract contained less than 100 words. For the latter predicate, we need a regular expression:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza" &&
                     wordCounter.Matches (article.Abstract).Count < 100);
```



The problem is that SQL Server doesn't support regular expressions, so the LINQ-to-db providers will throw an exception, complaining that the query cannot be translated to SQL. We can solve this by querying in two steps: first retrieving all articles on influenza through a LINQ to SQL query, and then filtering *locally* for abstracts of less than 100 words:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

IEnumerable<MedicalArticle> sqlQuery = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza");

IEnumerable<MedicalArticle> localQuery = sqlQuery
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Because `sqlQuery` is of type `IEnumerable<MedicalArticle>`, the second query binds to the local query operators, forcing that part of the filtering to run on the client.

With `AsEnumerable`, we can do the same in a single query:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");

var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza")
.AsEnumerable()
.Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

An alternative to calling `AsEnumerable` is to call `ToArray` or `ToList`. The advantage of `AsEnumerable` is that it doesn't force immediate query execution, nor does it create any storage structure.

Moving query processing from the database server to the client can hurt performance, especially if it means retrieving more

can hurt performance, especially if it means retrieving more rows. A more efficient (though more complex) way to solve our example would be to use SQL CLR integration to expose a function on the database that implemented the regular expression.

We demonstrate combined interpreted and local queries further in Chapter 10.

LINQ to SQL and Entity Framework

Throughout this and the following chapter, we use LINQ to SQL (L2S) and Entity Framework (EF) to demonstrate interpreted queries. We'll now examine the key features of these technologies.

If you're already familiar with L2S, take an advance look at Table 8-1 (at end of this section) for a summary of the API differences with respect to querying.

LINQ to SQL Versus Entity Framework

Both LINQ to SQL and Entity Framework are LINQ-enabled object-relational mappers. The essential difference is that EF allows for stronger decoupling between the database schema and the classes that you query. Instead of querying classes that closely represent the database schema, you query a higher-level abstraction described by an *Entity Data Model*. This offers extra flexibility, but incurs a cost in both performance and simplicity.

L2S was written by the C# team and was released with Framework 3.5; EF was written by the ADO.NET team and was released later as part of Service Pack 1. L2S has since been taken over by the ADO.NET team. This has resulted in the product receiving only minor improvements in Framework 4.0, with the team concentrating more on EF.

EF has improved considerably in Framework 4.0, although each technology still has unique strengths. L2S's strengths are ease of use, simplicity, performance, and the quality of its SQL translations. EF's strength is its flexibility in creating sophisticated mappings between the database and entity classes. EF also allows for databases other than SQL Server via a *provider model* (L2S also features a provider

databases other than SQL Server via a *provider model* (L2S also features a provider model, but this was made internal to encourage third parties to focus on EF instead).

A welcome enhancement in EF 4.0 is that it now supports (almost) the same querying functionality as L2S. This means that the LINQ-to-db queries that we demonstrate in this book work with either technology. Further, it makes L2S excellent for learning how to query databases in LINQ—because it keeps the object-relational side of things simple while you learn querying principles that also work with EF.

LINQ to SQL Entity Classes

L2S allows you to use any class to represent data, as long as you decorate it with appropriate attributes. Here's a simple example:

```
[Table]
public class Customer
{
  [Column(IsPrimaryKey=true)]
  public int ID;
```

```
    public int ID;
```

```
    [Column]
    public string Name;
}
```

The [Table] attribute, in the System.Data.Linq.Mapping namespace, tells L2S that an object of this type represents a row in a database table. By default, it assumes the table name matches the class name; if this is not the case, you can specify the table name as follows:

```
[Table (Name="Customers")]
```

A class decorated with the [Table] attribute is called an *entity* in L2S. To be useful, its structure must closely—or exactly—match that of a database table, making it a low-level construct.

The [Column] attribute flags a field or property that maps to a column in a table. If the column name differs from the field or property name, you can specify the column name as follows:

```
[Column (Name="FullName")]
public string Name;
```

```
[Column (Name = "LastName")]
public string Name;
```

The `IsPrimaryKey` property in the `[Column]` attribute indicates that the column partakes in the table's primary key and is required for maintaining object identity, as well as allowing updates to be written back to the database.

Instead of defining public fields, you can define public properties in conjunction with private fields. This allows you to write validation logic into the property accessors. If you take this route, you can optionally instruct L2S to bypass your property accessors and write to the field directly when populating from the database:

```
string _name;

[Column (Storage="_name")]
public string Name { get { return _name; } set { _name = value; } }
```

`Column(Storage="_name")` tells L2S to write directly to the `_name` field (rather than the `Name` property) when populating the entity. L2S's use of reflection allows the field to be private—as in this example.

be private—as in this example.

You can generate entity classes automatically from a database using either Visual Studio (add a new "LINQ to SQL Classes" project item) or with the *SqlMetal* command-line tool.

Entity Framework Entity Classes

As with L2S, EF lets you use any class to represent data (although you have to implement special interfaces if you want functionality such as navigation properties).

The following entity class, for instance, represents a customer that ultimately maps to a *customer* table in the database:

```
// You'll need to reference System.Data.Entity.dll

[EdmEntityType (NamespaceName = "NutshellModel", Name = "Customer")]
public partial class Customer
{
```

```
public partial class Customer
{
  [EdmScalarPropertyAttribute (EntityKeyProperty = true, IsNullable = false)]
  public int ID { get; set; }

  [EdmScalarProperty (EntityKeyProperty = false, IsNullable = false)]
  public string Name { get; set; }
}
```


Unlike with L2S, however, a class such as this is not enough on its own. Remember that with EF, you're not querying the database directly—you're querying a higher-level model called the *Entity Data Model* (EDM). There needs to be some way to describe the EDM, and this is most commonly done via an XML file with an *.edmx* extension, which contains three parts:

-
-
-

The *conceptual model*, which describes the EDM in isolation of the database

The *conceptual model*, which describes the EDM in isolation of the database

The *store model*, which describes the database schema

The *mapping*, which describes how the conceptual model maps to the store

The easiest way to create an *.edmx* file is to add an "ADO.NET Entity Data Model" project item in Visual Studio and then follow the wizard for generating entities from a database. This creates not only the *.edmx* file, but the entity classes as well.

The entity classes in EF map to the *conceptual model*. The types that support querying and updating the conceptual model are collectively called *Object Services*.

The designer assumes that you initially want a simple 1:1 mapping between tables and entities. You can enrich this, however, by tweaking the EDM either with the designer or by editing the underlying *.edmx* file that it creates for you. Here are some of the things you can do:

-
-

-
-

Map several tables into one entity.

Map one table into several entities.

Map inherited types to tables using the three standard kinds of strategies popular in the ORM world.

The three kinds of inheritance strategies are:

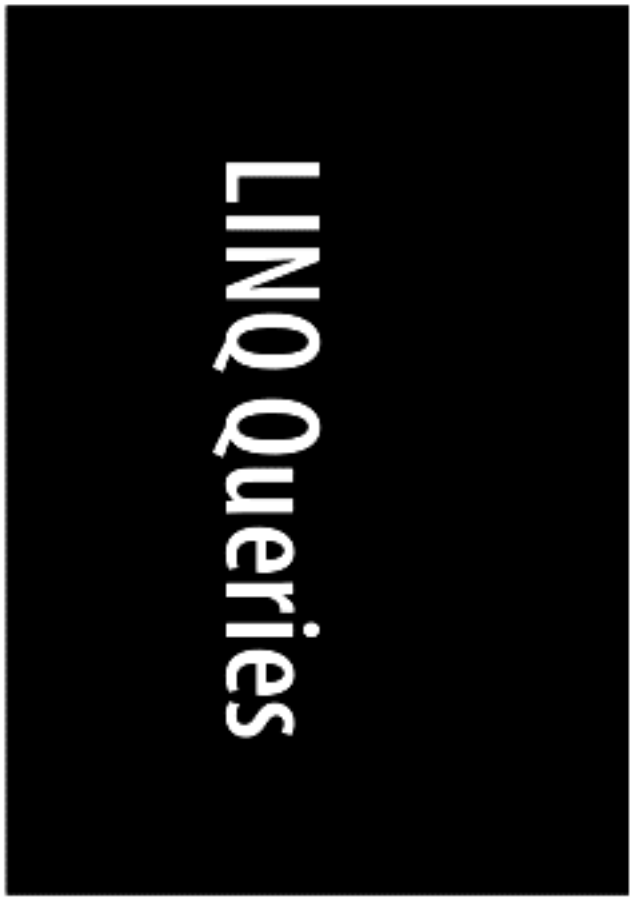
Table per hierarchy
: A single table maps to a whole class hierarchy. The table contains a discriminator column to indicate which type each row should map to.

Table per type
: A single table maps to one type, meaning that an inherited type maps to several tables. EF generates a SQL JOIN when you query an entity, to merge all its base types together.

Table per concrete type
: A separate table maps to each concrete type. This means that a base type maps to several tables and EF generates a SQL UNION when you query for entities

to several tables and EF generates a SQL UNION when you query for entities of a base type.

(In contrast, L2S supports only table per hierarchy.)



The EDM is complex: a thorough discussion can fill hundreds of pages! A good book that describes this in detail is Julia Lerman's *Programming Entity Framework*.

EF also lets you query through the EDM without LINQ—using a textual language called Entity SQL (ESQL). This can be useful for dynamically constructed queries.

DataContext and ObjectContext

Once you've defined entity classes (and an EDM in the case of EF), you can start querying. The first step is to instantiate a `DataContext` (L2S) or `ObjectContext` (EF), specifying a connection string:

```
var l2sContext = new DataContext ("database connection string");
var efContext = new ObjectContext ("entity connection string");
```

Instantiating a `DataContext`/`ObjectContext` directly is a low-level approach and is good for demonstrating how the classes work.

Instantiating a `DataContext`/`ObjectContext` directly is a low-level approach and is good for demonstrating how the classes work. More typically, though, you instantiate a *typed context* (a subclassed version of these classes), a process we'll describe shortly.

With L2S, you pass in the database connection string; with EF, you must pass an *entity connection string*, which incorporates the database connection string plus information on how to find the EDM. (If you've created an EDM in Visual Studio, you can find the entity connection string for your EDM in the *app.config* file.)

You can then obtain a queryable object calling `GetTable` (L2S) or `CreateObjectSet` (EF). The following example uses the `Customer` class that we defined earlier:

```
var context = new DataContext ("database connection string");
Table<Customer> customers = context.GetTable <Customer>();
```

```
Console.WriteLine (customers.Count());
Customer cust = customers.Single (c => c.ID == 2);
```

```
// # of rows in table.
// Retrieves Customer
```

```
// Retrieves Customer
// with ID of 2.
```

Here's the same thing with EF:

```
var context = new ObjectContext ("entity connection string");
context.DefaultContainerName = "NutshellEntities";
ObjectSet<Customer> customers = context.CreateObjectSet<Customer>();

Console.WriteLine (customers.Count());

Customer cust = customers.Single (c => c.ID == 2);

// # of rows in table.
// Retrieves Customer
// with ID of 2.
```

The `Single` operator is ideal for retrieving a row by primary key. Unlike `First`, it throws an exception if more than one element is returned.

A `DataContext`/`ObjectContext` object does two things. First, it acts as a factory for generating objects that you can query. Second, it keeps track of any changes that you make to your entities so that you can write them back. We can continue our previous example to update a customer with L2S as follows:

```
Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SubmitChanges();
```

With EF, the only difference is that you call `SaveChanges` instead:

```
Customer cust = customers.OrderBy (c => c.Name).First();
cust.Name = "Updated Name";
context.SaveChanges();
```

```
context.SaveChanges();
```

Typed contexts

Having to call `GetTable<Customer>()` or `CreateObjectSet<Customer>()` all the time is awkward. A better approach is to subclass `DataContext`/`ObjectContext` for a particular database, adding properties that do this for each entity. This is called a *typed context*:

```
class NutshellContext : DataContext    // For LINQ to SQL
{
  public Table<Customer> Customers
  {
    get { return GetTable<Customer>(); }
  }
  // ... and so on, for each table in the database
}
```

Here's the same thing for EF:

Here's the same thing for EF:

```
class NutshellContext : ObjectContext   // For Entity Framework
{
  public ObjectSet<Customer> Customers
  {
    get { return CreateObjectSet<Customer>(); }
  }
  // ... and so on, for each entity in the conceptual model
}
```

You can then simply do this:

```
var context = new NutshellContext ("connection string");
Console.WriteLine (context.Customers.Count());
```

If you use Visual Studio to create a "LINQ to SQL Classes" or "ADO.NET Entity Data Model" project item, it builds a typed context for you automatically. The designers can also do additional work such as pluralizing identifiers—in this example,

it's `context.Customers` and not `context.Customer`, even though the SQL table and entity class are both called `Customer`.

Disposing DataContext/ObjectContext

Although `DataContext`/`ObjectContext` implement `IDisposable`, you can (in general) get away without disposing instances. Disposing forces the context's connection to dispose—but this is usually unnecessary because L2S and EF close connections

get away without disposing instances. Disposing forces the context's connection to dispose—but this is usually unnecessary because L2S and EF close connections automatically whenever you finish retrieving results from a query.

Disposing a context can actually be problematic because of lazy evaluation. Consider the following:

```
IQueryable<Customer> GetCustomers (string prefix)
{
  using (var dc = new NutshellContext ("connection string"))
    return dc.GetTable<Customer>()
             .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
  Console.WriteLine (c.Name);
```

This will fail because the query is evaluated when we enumerate it—which is *after* disposing its `DataContext`.

There are some caveats, though, on not disposing contexts:

- It relies on the connection object releasing all unmanaged resources on the `Close` method. While this holds true with `SqlConnection`, it's theoretically possible for a third-party connection to keep resources open if you call `Close` but not `Dispose` (though this would arguably violate the contract defined by `IDbConnection.Close`).

fined by `IDbConnection.Close`).

- If you manually call `GetEnumerator` on a query (instead of using `foreach`) and then fail to either dispose the enumerator or consume the sequence, the connection will remain open. Disposing the `DataContext`/`ObjectContext` provides a backup in such scenarios.
- Some people feel that it's tidier to dispose contexts (and all objects that implement `IDisposable`).

If you want to explicitly dispose contexts, you must pass a `DataContext`/`ObjectContext` instance into methods such as `GetCustomers` to avoid the problem described.

Object tracking

A `DataContext`/`ObjectContext` instance keeps track of all the entities it instantiates, so it can feed the same ones back to you whenever you request the same rows in a table. In other words, a context in its lifetime will never emit two separate entities that refer to the same row in a table (where a row is identified by primary key).

You can disable this behavior in L2S by setting `ObjectTrackingEnabled` to `false` on the `DataContext` object. In EF, you can disable change tracking on a per-type basis:

```
context.Customers.MergeOption = MergeOption.NoTracking;
```

Disabling object tracking also prevents you from submitting updates to the data.

To illustrate object tracking, suppose the customer whose name is alphabetically first also has the lowest ID. In the following example, `a` and `b` will reference the same object:

```
var context = new NutshellContext ("connection string");

Customer a = context.Customers.OrderBy (c => c.Name).First();
Customer b = context.Customers.OrderBy (c => c.ID).First();
```

This has a couple of interesting consequences. First, consider what happens when

This has a couple of interesting consequences. First, consider what happens when L2S or EF encounters the second query. It starts by querying the database—and obtaining a single row. It then reads the primary key of this row and performs a lookup in the context's entity cache. Seeing a match, it returns the existing object *without updating any values*. So, if another user had just updated that customer's `Name` in the database, the new value would be ignored. This is essential for avoiding unexpected side effects (the `Customer` object could be in use elsewhere) and also for managing concurrency. If you had altered properties on the `Customer` object and not yet called `SubmitChanges`/`SaveChanges`, you wouldn't want your properties automatically overwritten.

To get fresh information from the database, you must either instantiate a new context or call its `Refresh` method, passing in the entity or entities that you want refreshed.

The second consequence is that you cannot explicitly project into an entity type—to select a subset of the row's columns—without causing trouble. For example, if you want to retrieve only a customer's name, any of the following approaches is valid:

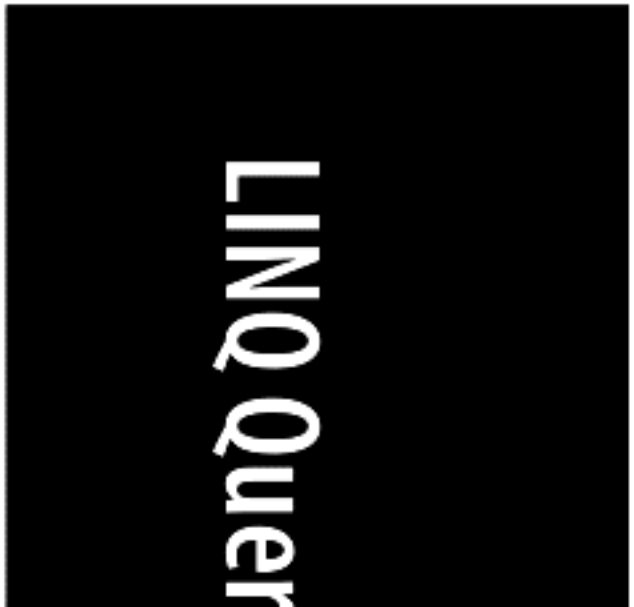
valid:

```
customers.Select (c => c.Name);
customers.Select (c => new { Name = c.Name } );
customers.Select (c => new MyCustomType { Name = c.Name } );
```

The following, however, is not:

```
customers.Select (c => new Customer { Name = c.Name } );
```

This is because the `Customer` entities will end up partially populated. So, the next time you perform a query that requests *all* customer columns, you get the same cached `Customer` objects with only the `Name` property populated.



In a multitier application, you cannot use a single static instance of a `DataContext` or `ObjectContext` in the middle tier to handle all requests, because contexts are not thread-safe. Instead, middle-tier methods must create a fresh context per client request. This is actually beneficial because it shifts the burden in handling simultaneous updates to the database server, which is properly equipped for the job. A database server, for instance, will apply transaction isolation-level semantics.

Associations

Associations

The entity generation tools perform another useful job. For each relationship defined in your database, they generate properties on each side that allow you to query that relationship. For example, suppose we define a customer and purchase table in a one-to-many relationship:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
```

```
create table Purchase
(
  ID int not null primary key,
```

```
    ID int not null primary key,
    CustomerID int references Customer (ID),
    Description varchar(30) not null,
    Price decimal not null
  )
```

With automatically generated entity classes, we can write queries such as this:

```
var context = new NutshellContext ("connection string");

// Retrieve all purchases made by the first customer (alphabetically):

Customer cust1 = context.Customers.OrderBy (c => c.Name).First();
foreach (Purchase p in cust1.Purchases)
  Console.WriteLine (p.Price);

// Retrieve the customer who made the lowest value purchase:

Purchase cheapest = context.Purchases.OrderBy (p => p.Price).First();
```

```
Purchase cheapest = context.Purchases.OrderBy (p => p.Price).First();
Customer cust2 = cheapest.Customer;
```

Further, if `cust1` and `cust2` happened to refer to the same customer, `c1` and `c2` would *refer to the same object*: `cust1==cust2` would return `true`.

Let's examine the signature of the automatically generated `Purchases` property on the `Customer` entity. With L2S:

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```


With EF:

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK...", "Purchase")]
public EntityCollection<Purchase> Purchases { get {...} set {...} }
```

An `EntitySet` or `EntityCollection` is like a predefined query, with a built-in `Where` clause that extracts related entities. The `[Association]` attribute gives L2S the information it needs to formulate the SQL query; the `[EdmRelationshipNavigation`

clause that extracts related entities. The [Association] attribute gives L2S the information it needs to formulate the SQL query; the [EdmRelationshipNavigationProperty] attribute tells EF where to look in the EDM for information about that relationship.

As with any other type of query, you get deferred execution. With L2S, an EntitySet is populated when you enumerate over it; with EF, an EntityCollection is populated when you explicitly call its Load method.

Here's the Purchases.Customer property, on the other side of the relationship, with L2S:

```
[Association (Storage="_Customer",ThisKey="CustomerID",IsForeignKey=true)]
public Customer Customer { get {...} set {...} }
```

Although the property is of type Customer, its underlying field (_Customer) is of type EntityRef. The EntityRef type implements deferred loading, so the related Customer is not retrieved from the database until you actually ask for it.

EF works in the same way, except that it doesn't populate the property simply by you accessing it: you must call Load on its EntityReference object. This means EF contexts must expose properties for both the actual parent object and its EntityReference wrapper:

ference wrapper:

```
[EdmRelationshipNavigationProperty ("NutshellModel", "FK..., "Customer")]
public Customer Customer { get {...} set {...} }
```

```
public EntityReference<Customer> CustomerReference { get; set; }
```

You can make EF behave like L2S and have it populate `EntityCollections` and `EntityReferences` simply by virtue of their properties being accessed as follows:

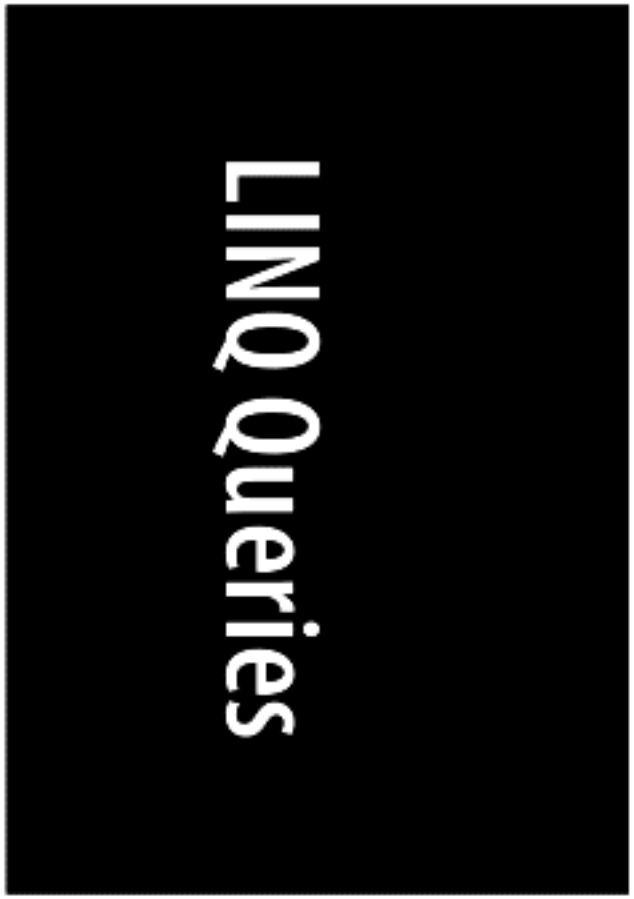
```
context.ContextOptions.DeferredLoadingEnabled = true;
```

Deferred Execution with L2S and EF

L2S and EF queries are subject to deferred execution, just like local queries. This allows you to build queries progressively. There is one aspect, however, in which L2S/EF have special deferred execution semantics, and that is when a subquery appears inside a `Select` expression:

-

With local queries, you get double deferred execution, because from a functional perspective, you're selecting a sequence of *queries*. So, if you enumerate the outer result sequence, but never enumerate the inner sequences, the subquery will never execute.



- With L2S/EF, the subquery is executed at the same time as the main outer query. This avoids excessive round-tripping.

For example, the following query executes in a single round trip upon reaching the first `foreach` statement:

```
var context = new NutshellContext ("connection string");

var query = from c in context.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine (namePrice.Name + " spent " + namePrice.Price);
```

```
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine (namePrice.Name + " spent " + namePrice.Price);
```

Any EntitySets/EntityCollections that you explicitly project are fully populated in a single round trip:

```
var query = from c in context.Customers
            select new { c.Name, c.Purchases };

foreach (var row in query)
  foreach (Purchase p in row.Purchases)   // No extra round-tripping
    Console.WriteLine (row.Name + " spent " + p.Price);
```

But if we enumerate EntitySet/EntityCollection properties without first having projected, deferred execution rules apply. In the following example, L2S and EF execute another Purchases query on each loop iteration:

```
context.ContextOptions.DeferredLoadingEnabled = true;  // For EF only.

foreach (Customer c in context.Customers)
  foreach (Purchase p in c.Purchases)    // Another SQL round-trip
    Console.WriteLine (c.Name + " spent " + p.Price);
```

```
    Console.WriteLine (c.Name + " spent " + p.Price);
```

This model is advantageous when you want to *selectively* execute the inner loop, based on a test that can be performed only on the client:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    foreach (Purchase p in c.Purchases)   // Another SQL round trip
      Console.WriteLine (...);
```

(In Chapter 9, we will explore `Select` subqueries in more detail, in "Projecting" on page 369.)

We've seen that you can avoid round-tripping by explicitly projecting associations. L2S and EF offer other mechanisms for this too, which we cover in the following two sections.

DataLoadOptions

The `DataLoadOptions` class is specific to L2S. It has two distinct uses:

- It lets you specify, in advance, a filter for `EntitySet` associations (`AssociateWith`).

- It lets you request that certain `EntitySets` be eagerly loaded, to lessen round-tripping (`LoadWith`).

Specifying a filter in advance

Let's refactor our previous example as follows:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    ProcessCustomer (c);
```

We'll define `ProcessCustomer` like this:

We'll define ProcessCustomer like this:

```
void ProcessCustomer (Customer c)
{
  Console.WriteLine (c.ID + " " + c.Name);
  foreach (Purchase p in c.Purchases)
    Console.WriteLine ("  - purchased a " + p.Description);
}
```

Now suppose we want to feed ProcessCustomer only a *subset* of each customer's purchases; say, the high-value ones. Here's one solution:

```
foreach (Customer c in context.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
    ProcessCustomer (c.ID,
                     c.Name,
                     c.Purchases.Where (p => p.Price > 1000));
...
void ProcessCustomer (int custID, string custName,
                      IEnumerable<Purchase> purchases)
{
```

```
{
  Console.WriteLine (custID + " " + custName);
  foreach (Purchase p in purchases)
    Console.WriteLine ("  - purchased a " + p.Description);
}
```

This is messy. It would get messier still if `ProcessCustomer` required more `Customer` fields. A better solution is to use `DataLoadOptions`'s `AssociateWith` method:

```
DataLoadOptions options = new DataLoadOptions();
options.AssociateWith <Customer>
 (c => c.Purchases.Where (p => p.Price > 1000));
context.LoadOptions = options;
```

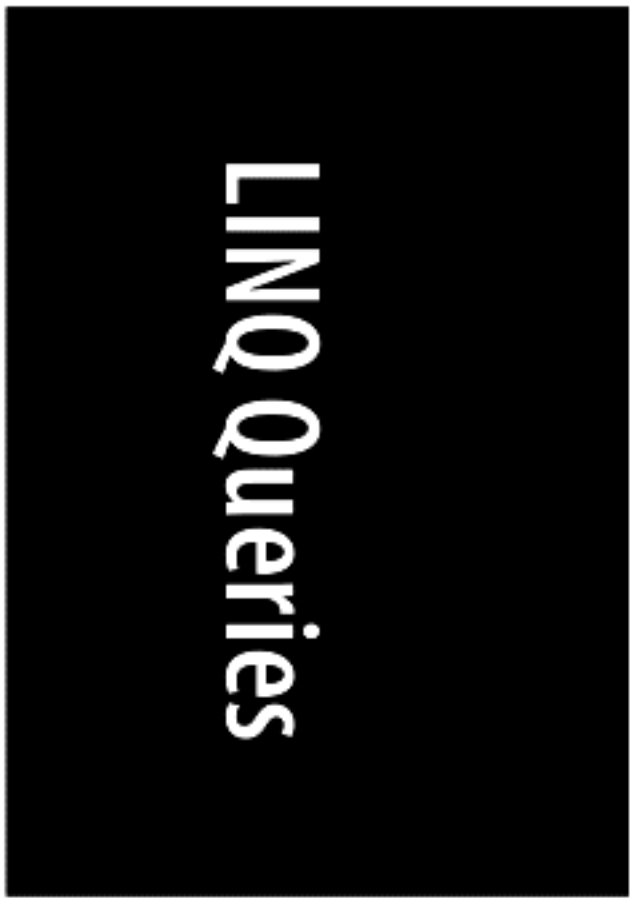
This instructs our `DataContext` instance always to filter a `Customer`'s `Purchases` using the given predicate. We can now use the original version of `ProcessCustomer`.

`AssociateWith` doesn't change deferred execution semantics. When a particular relationship is used, it simply instructs to implicitly add a particular filter to the equation.

Eager loading

Eager loading

The second use for a `DataLoadOptions` is to request that certain `EntitySet`s be eagerly loaded with their parent. For instance, suppose you want to load all customers and their purchases in a single SQL round trip:



```
DataLoadOptions options = new DataLoadOptions();
options.LoadWith <Customer> (c => c.Purchases);
context.LoadOptions = options;

foreach (Customer c in context.Customers)     // One round trip:
  foreach (Purchase p in c.Purchases)
    Console.WriteLine (c.Name + " bought a " + p.Description);
```

This instructs that whenever a `Customer` is retrieved, its `Purchases` should also be retrieved at the same time. You can combine `LoadWith` with `AssociateWith`. The following instructs that whenever a customer is retrieved, its *high-value* purchases should be retrieved in the same round trip:

```
options.LoadWith <Customer> (c => c.Purchases);
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
```

Eager Loading in Entity Framework

You can request in EF that associations be eagerly loaded with the `Include` method. The following enumerates over each customer's purchases—while generating just one SQL query:

```
foreach (var c in context.Customers.Include ("Purchases"))
  foreach (var p in c.Purchases)
    Console.WriteLine (p.Description);
```

`Include` can be used with arbitrary breadth and depth. For example, if each `Purchase` also had `PurchaseDetails` and `SalesPersons` navigation properties, the entire nested structure could be eagerly loaded as follows:

```
context.Customers.Include ("Purchases.PurchaseDetails")
                 .Include ("Purchases.SalesPersons")
```

Updates

L2S and EF also keep track of changes that you make to your entities and allow you to write them back to the database by calling `SubmitChanges` on the `DataContext` object, or `SaveChanges` on the `ObjectContext` object.

object, or SaveChanges on the ObjectContext object.

L2S's Table<> class provides InsertOnSubmit and DeleteOnSubmit methods for inserting and deleting rows in a table; EF's ObjectSet<> class provides AddObject and DeleteObject methods to do the same thing. Here's how to insert a row:

```
var context = new NutshellContext ("connection string");

Customer cust = new Customer { ID=1000, Name="Bloggs" };
context.Customers.InsertOnSubmit (cust);   // AddObject with EF
context.SubmitChanges();                   // SaveChanges with EF
```

We can later retrieve that row, update it, and then delete it:

```
var context = new NutshellContext ("connection string");

Customer cust = context.Customers.Single (c => c.ID == 1000);
cust.Name = "Bloggs2";
```



```
context.SubmitChanges();
```

```
context.SubmitChanges();

// Updates the customer
// DeleteObject with EF
// Deletes the customer

context.Customers.DeleteOnSubmit (cust);
context.SubmitChanges();
```

`SubmitChanges`/`SaveChanges` gathers all the changes that were made to its entities since the context's creation (or the last save), and then executes a SQL statement to write them to the database. Any `TransactionScope` is honored; if none is present, it wraps all statements in a new transaction.

You can also add new or existing rows to an `EntitySet`/`EntityCollection` by calling `Add`. L2S and EF automatically populate the foreign keys when you do this (after calling `SubmitChanges` or `SaveChanges`):

Add. L2S and EF automatically populate the foreign keys when you do this (after calling SubmitChanges or SaveChanges):

```
Purchase p1 = new Purchase { ID=100, Description="Bike",  Price=500 };
Purchase p2 = new Purchase { ID=101, Description="Tools", Price=100 };

Customer cust = context.Customers.Single (c => c.ID == 1);

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);
context.SubmitChanges();

//

(or SaveChanges with EF)
```

If you don't want the burden of allocating unique keys, you can

If you don't want the burden of allocating unique keys, you can use either an auto-incrementing field (IDENTITY in SQL Server) or a `Guid` for the primary key.

In this example, L2S/EF automatically writes `1` into the `CustomerID` column of each of the new purchases (L2S knows to do this because of the association attribute that we defined on the `Purchases` property; EF knows to do this because of information in the EDM).

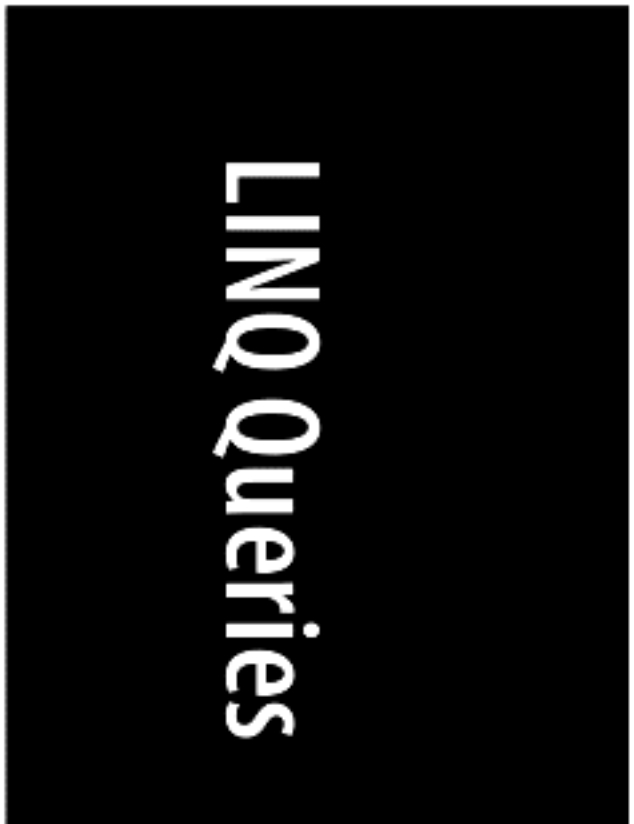
```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

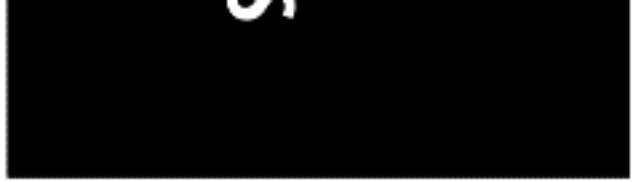
If the `Customer` and `Purchase` entities were generated by the Visual Studio designer or the *SqlMetal* command-line tool, the generated classes would include further code to keep the two sides of each relationship in sync. In other words, assigning the `Purchase.Customer` property would automatically add the new customer to the `Customer.Purchases` entity set—and vice versa. We can illustrate this by rewriting the preceding example as follows:

```
var context = new NutshellContext ("connection string");

Customer cust = context.Customers.Single (c => c.ID == 1);
```

```
Customer cust = context.Customers.Single (c => c.ID == 1);
new Purchase { ID=100, Description="Bike",  Price=500, Customer=cust };
new Purchase { ID=101, Description="Tools", Price=100, Customer=cust };
```

```
context.SubmitChanges();

// (SaveChanges with EF)
```





When you remove a row from an EntitySet/EntityCollection, its foreign key field is automatically set to null. The following disassociates our two recently added purchases from their customer:

```
var context = new NutshellContext ("connection string");

Customer cust = context.Customers.Single (c => c.ID == 1);

cust.Purchases.Remove (cust.Purchases.Single (p => p.ID == 100));
cust.Purchases.Remove (cust.Purchases.Single (p => p.ID == 101));

context.SubmitChanges();

// Submit SQL to database (SaveChanges in EF)
```

Because this tries to set each purchase's `CustomerID` field to `null`, `Purchase.CustomerID` must be nullable in the database; otherwise, an exception is thrown. (Further, the `CustomerID` field or property in the entity class must be a nullable type.)

To delete child entities entirely, remove them from the `Table<>` or `ObjectSet<>` instead. With L2S:

```
var c = context;
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 100));
c.Purchases.DeleteOnSubmit (c.Purchases.Single (p => p.ID == 101));
c.SubmitChanges();          // Submit SQL to database
```

With EF:

```
var c = context;
c.Purchases.DeleteObject (c.Purchases.Single (p => p.ID == 100));
c.Purchases.DeleteObject (c.Purchases.Single (p => p.ID == 101));
c.SaveChanges();          // Submit SQL to database
```

API Differences Between L2S and EF

API Differences Between L2S and EF

As we've seen, L2S and EF are similar in the aspect of querying with LINQ and performing updates. Table 8-1 summarizes the API differences.

Table 8-1. API differences between L2S and EF

Purpose	LINQ to SQL	Entity Framework
Gatekeeper class for all CRUD operations	DataContext	ObjectContext
Method to (lazily) retrieve all entities of a given type from the store	GetTable	CreateObjectSet

Type returned by the above method

Method to update the store with any additions, modifications, or deletions to entity objects

Table<T>

SubmitChanges

SubmitChanges

InsertOnSubmit

DeleteOnSubmit

EntitySet<T>

ObjectSet<T>

SaveChanges

AddObject

DeleteObject

EntityCollection<T>

Method to add a new entity to the store when the context is

Method to add a new entity to the store when the context is updated

Method to delete an entity from the store when the context is updated

Type to represent one side of a relationship property, when that side has a multiplicity of many

Purpose	LINQ to SQL	Entity Framework
Type to represent one side of a relationship property, when that side has a multiplicity of one	`EntityRef<T>`	`EntityReference<T>`
Default strategy for loading relationship properties	Lazy	Explicit
Construct that enables eager loading	`DataLoadOptions`	`.Include()`

Building Query Expressions

Building Query Expressions

So far in this chapter, when we've needed to dynamically compose queries, we've done so by conditionally chaining query operators. Although this is adequate in many scenarios, sometimes you need to work at a more granular level and dynamically compose the lambda expressions that feed the operators.

In this section, we'll assume the following `Product` class:

```
[Table] public partial class Product
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Description;
  [Column]                    public bool Discontinued;
  [Column]                    public DateTime LastSale;
}
```

Delegates Versus Expression Trees

Recall that:

Recall that:

-
-

Local queries, which use `Enumerable` operators, take delegates.

Interpreted queries, which use `Queryable` operators, take expression trees.

We can see this by comparing the signature of the `Where` operator in `Enumerable` and `Queryable`:

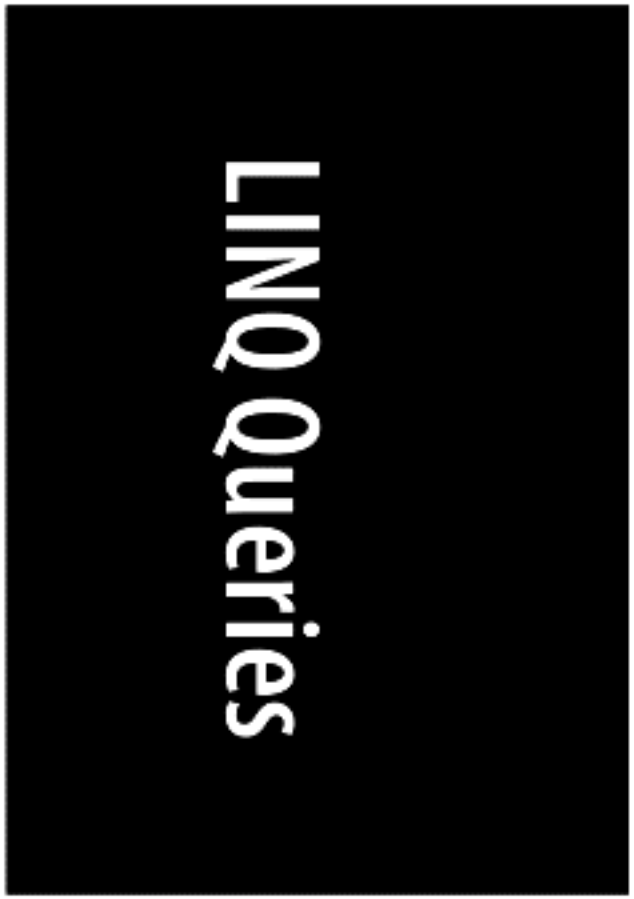
```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

```
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool>> predicate)
```

When embedded within a query, a lambda expression looks identical whether it binds to `Enumerable`'s operators or `Queryable`'s operators:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

When you assign a lambda expression to an intermediate variable, however, you must be explicit on whether to resolve to a delegate (i.e., `Func<>`) or an expression tree (i.e., `Expression<Func<>>`). In the following example, `predicate1` and `predicate2` are not interchangeable:



```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

Compiling expression trees

You can convert an expression tree to a delegate by calling `Compile`. This is of particular value when writing methods that return reusable expressions. To illustrate, we'll add a static method to the `Product` class that returns a predicate evaluating to `true` if a product is not discontinued and has sold in the past 30 days:

```
public partial class Product
{
  public static Expression<Func<Product, bool>> IsSelling()
  {
    return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
  }
}
```

```
  }
}
```

(We've defined this in a separate partial class to avoid being overwritten by an automatic `DataContext` generator such as Visual Studio's code generator.)

The method just written can be used both in interpreted and in local queries:

```
void Test()
{
  var dataContext = new NutshellContext ("connection string");
  Product[] localProducts = dataContext.Products.ToArray();

  IQueryable<Product> sqlQuery =
    dataContext.Products.Where (Product.IsSelling());

  IEnumerable<Product> localQuery =
    localProducts.Where (Product.IsSelling.Compile());
}
```

You cannot convert in the reverse direction, from a delegate to an expression tree. This makes expression trees more versatile.

You cannot convert in the reverse direction, from a delegate to an expression tree. This makes expression trees more versatile.

AsQueryable

The AsQueryable operator lets you write whole *queries* that can run over either local or remote sequences:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
  return from p in input
         where ...
         order by ...
         select p;
}
```



```
void Test()
{
  var dataContext = new NutshellContext ("connection string");
```

```
{
  var dataContext = new NutshellContext ("connection string");
  Product[] localProducts = dataContext.Products.ToArray();

  var sqlQuery   = FilterSortProducts (dataContext.Products);
  var localQuery = FilterSortProducts (localProducts.AsQueryable());
  ...
}
```

`AsQueryable` wraps `IQueryable<T>` clothing around a local sequence so that subsequent query operators resolve to expression trees. When you later enumerate over the result, the expression trees are implicitly compiled (at a small performance cost), and the local sequence enumerates as it would ordinarily.

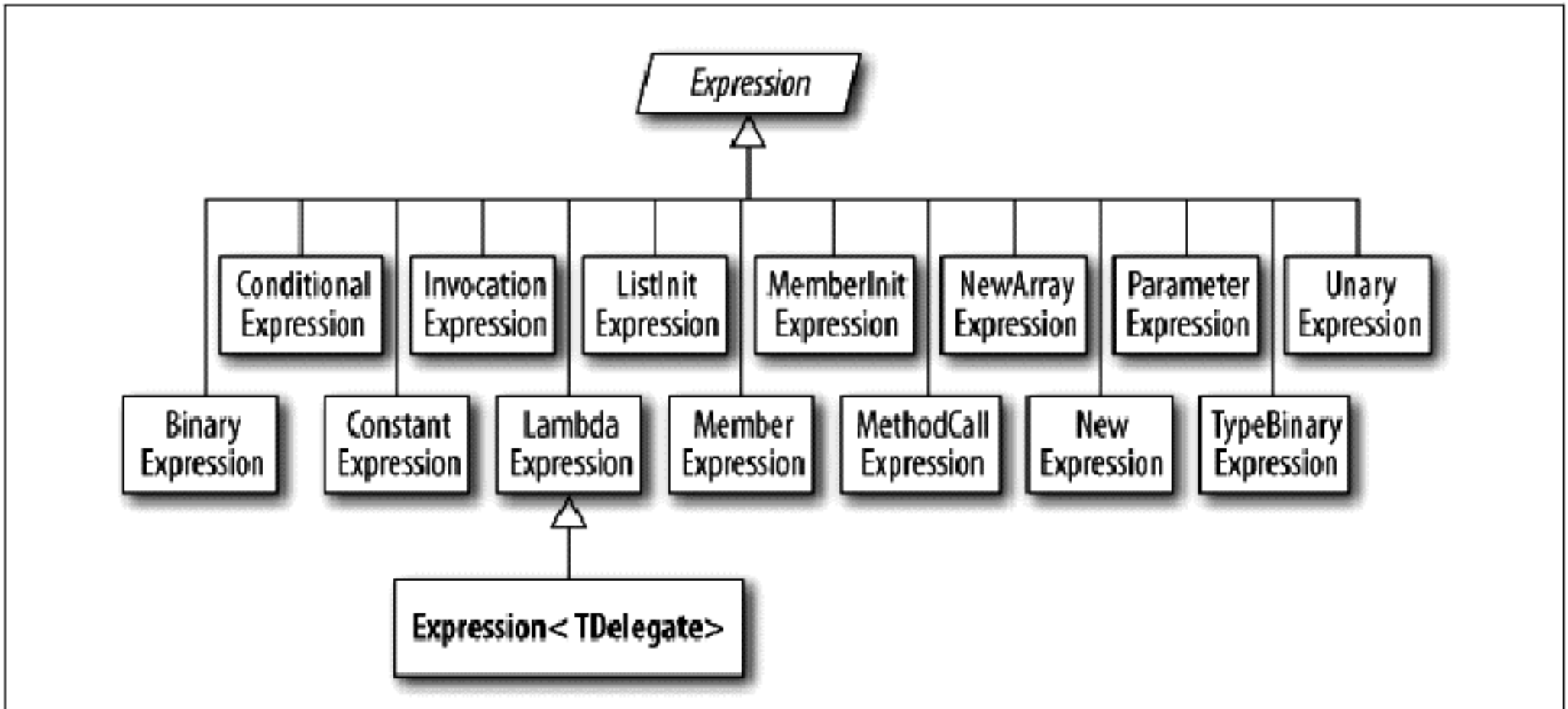
Expression Trees

We said previously, that assigning a lambda expression to a variable of type `Expression<TDelegate>` causes the C# compiler to emit an expression tree. With some programming effort, you can do the same thing manually at runtime—in other words, dynamically build an expression tree from scratch. The result can be cast to an `Expression<TDelegate>` and used in LINQ-to-db queries, or compiled into an

words, dynamically build an expression tree from scratch. The result can be cast to an `Expression<TDelegate>` and used in LINQ-to-db queries, or compiled into an ordinary delegate by calling `Compile`.

The Expression DOM

An expression tree is a miniature code DOM. Each node in the tree is represented by a type in the `System.Linq.Expressions` namespace; these types are illustrated in Figure 8-11.



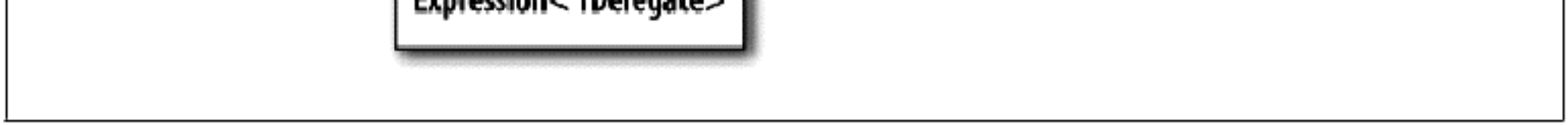


Figure 8-11. Expression types

From Framework 4.0, this namespace features additional ex-

From Framework 4.0, this namespace features additional expression types and methods to support language constructs that can appear in code blocks. These are for the benefit of the DLR and not lambda expressions. In other words, code-block-style lambdas still cannot be converted to expression trees:

```
Expression<Func<Customer,bool>> invalid =
  { return true; ]  // Code blocks not permitted
```

The base class for all nodes is the (nongeneric) `Expression` class. The generic `Expression<TDelegate>` class actually means "typed lambda expression" and might have been named `LambdaExpression<TDelegate>` if it wasn't for the clumsiness of this:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

`Expression<>`'s base type is the (nongeneric) `LambdaExpression` class. `LamdbaExpression` provides type unification for lambda expression trees: any typed `Expression<>` can be cast to a `LambdaExpression`.

The thing that distinguishes `LambdaExpressions` from ordinary `Expressions` is that lambda expressions have *parameters*.

lambda expressions have *parameters*.

To create an expression tree, don't instantiate node types directly; rather, call static methods provided on the `Expression` class. Here are all the methods:

```
Add
AddChecked
And
AndAlso
ArrayIndex
ArrayLength
Bind
Call
Coalesce
Condition
```

Condition
Constant
Convert
ConvertChecked
Divide

ElementInit
Equal
ExclusiveOr
Field
GreaterThan
GreaterThanOrEqual
Invoke

Invoke
Lambda
LeftShift
LessThan
LessThanOrEqual
ListBind
ListInit
MakeBinary
MakeMemberAccess
MakeUnary
MemberBind

MemberBind
MemberInit
Modulo
Multiply
MultiplyChecked
Negate
NegateChecked
New
NewArrayBounds
NewArrayInit
Not
NotEqual

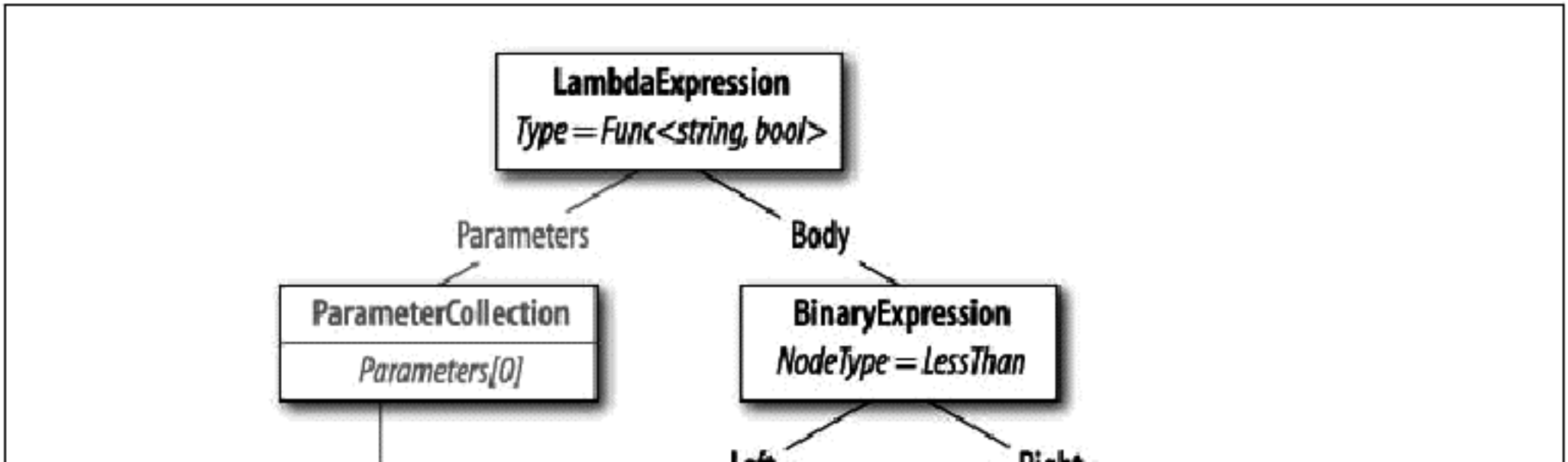
NotEqual

Or
OrElse
Parameter
Power
Property
PropertyOrField
Quote
RightShift
Subtract
SubtractChecked
TypeAs

```
TypeAs
TypeIs
UnaryPlus
```

Figure 8-12 shows the expression tree that the following assignment creates:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

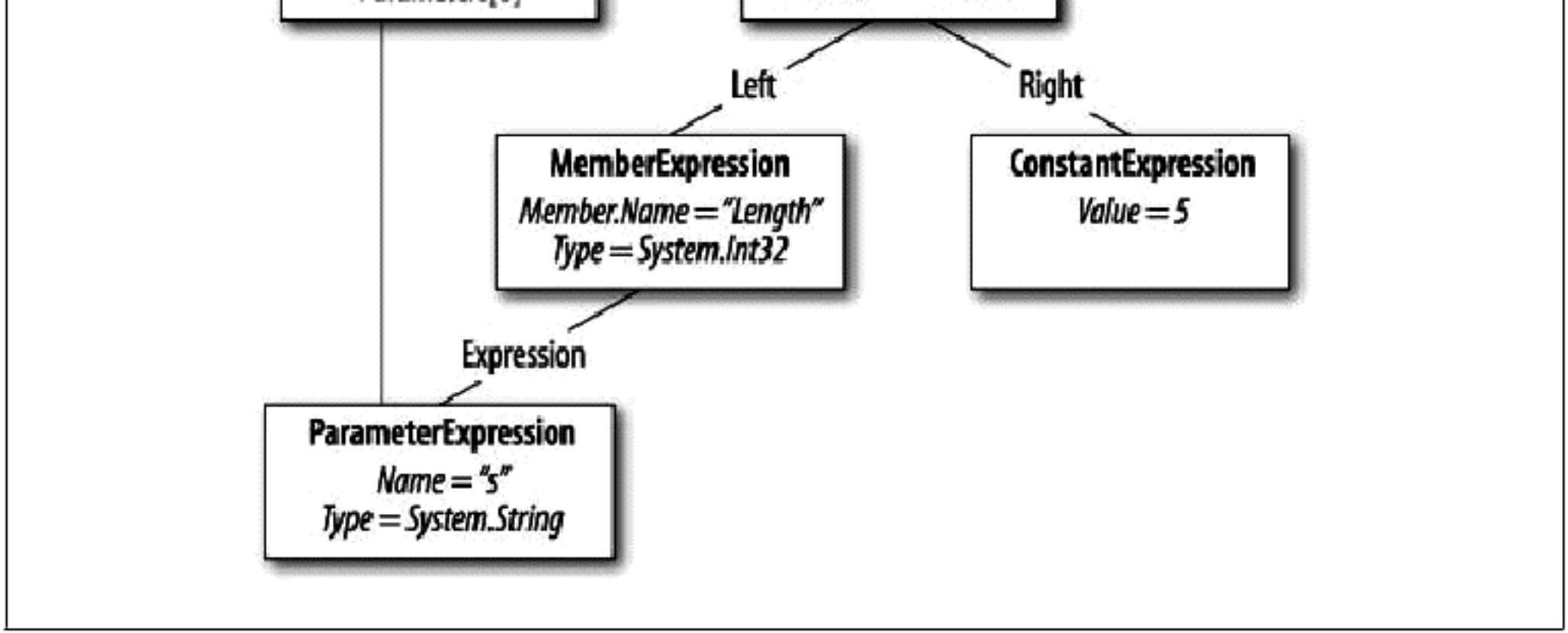


Figure 8-12. Expression tree

We can demonstrate this as follows:

```
Console.WriteLine (f.Body.NodeType);
Console.WriteLine (((BinaryExpression) f.Body).Right);

// LessThan
```

```
// LessThan
// 5
```

Let's now build this expression from scratch. The principle is that you start from the bottom of the tree and work your way up. The bottommost thing in our tree is a `ParameterExpression`, the lambda expression parameter called "s" of type `string`:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

The next step is to build the `MemberExpression` and `ConstantExpression`. In the former case, we need to access the `Length` *property* of our parameter, "s":

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Next is the `LessThan` comparison:

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

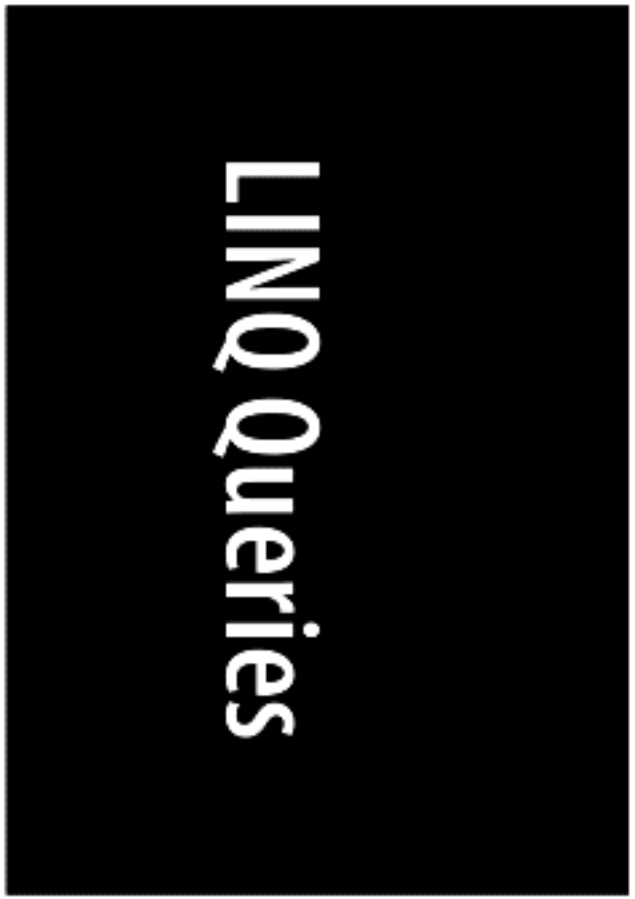
The final step is to construct the lambda expression, which links an expression `Body` to a collection of parameters:

```
Expression<Func<string, bool>> lambda
```

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

A convenient way to test our lambda is by compiling it to a delegate:

```
Func<string, bool> runnable = lambda.Compile();
```




```
Console.WriteLine (runnable ("kangaroo"));
```

```
Console.WriteLine (runnable ("kangaroo"));
Console.WriteLine (runnable ("dog"));

// False
// True
```


The easiest way to figure out which expression type to use is to examine an existing lambda expression in the Visual Studio debugger.

We continue this discussion online, at *http://www.albahari.com/expressions/*.

9

LINQ Operators

This chapter describes each of the LINQ query operators. As well as serving as a reference, two of the sections, "Projecting" on page 375 and "Joining" on page 370, cover a number of conceptual areas:

-
-
-

Projecting object hierarchies

Joining with `Select`, `SelectMany`, `Join`, and `GroupJoin`

Outer range variables in query expressions

All of the examples in this chapter assume that a `names` array is defined as follows:

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

Examples that query a database assume that a typed `DataContext` variable called `dataContext` is instantiated as follows:

```
var dataContext = new NutshellContext ("connection string...");
```

```
public class NutshellContext : DataContext
{
  public NutshellContext (string cxString) : base (cxString) {}
```

```
{
  public NutshellContext (string cxString) : base (cxString) {}

  public Table<Customer> Customers { get { return GetTable<Customer>(); } }
  public Table<Purchase> Purchases { get { return GetTable<Purchase>(); } }
}
```

```
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)]
  [Column]

public int ID;
public string Name;

  [Association (OtherKey="CustomerID")]
  public EntitySet<Purchase> Purchases = new EntitySet<Purchase>();
```

```
    public EntitySet<Purchase> Purchases = new EntitySet<Purchase>();
}
```

367

```
[Table] public class Purchase
{
    [Column(IsPrimaryKey=true)]
    [Column]
    [Column]
    [Column]
    [Column]
  EntityRef<Customer> custRef;
```

```
  EntityRef<Customer> custRef;

  public int ID;
  public int? CustomerID;
  public string Description;
  public decimal Price;
  public DateTime Date;

  [Association (Storage="custRef",ThisKey="CustomerID",IsForeignKey=true)]
  public Customer Customer
  {
    get { return custRef.Entity; } set { custRef.Entity = value; }
  }
}
```

All the examples in this chapter are preloaded into LINQPad, along with a sample database with a matching schema. You can

all the examples in this chapter are preloaded into LINQPad, along with a sample database with a matching schema. You can download LINQPad from *http://www.linqpad.net*.

The entity classes shown are a simplified version of what LINQ to SQL tools typically produce, and do not include code to update the opposing side in a relationship when their entities have been reassigned.

Here are the corresponding SQL table definitions:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
```

```
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

All examples will also work with Entity Framework, except where otherwise indicated. You can build an Entity Framework `ObjectContext` from these tables by creating a new Entity Data Model in Visual Studio, and then dragging the tables on to the designer surface.

Overview

Overview

In this section, we provide an overview of the standard query operators.

The standard query operators fall into three categories:

-
-
-

Sequence in, sequence out (sequence-to-sequence)

Sequence in, single element or scalar value out

Nothing in, sequence out (*generation* methods)

We first present each of the three categories and the query operators they include, and then we take up each individual query operator in detail.

Sequence→Sequence

Sequence→Sequence

Most query operators fall into this category—accepting one or more sequences as input and emitting a single output sequence. Figure 9-1 illustrates those operators that restructure the shape of the sequences.

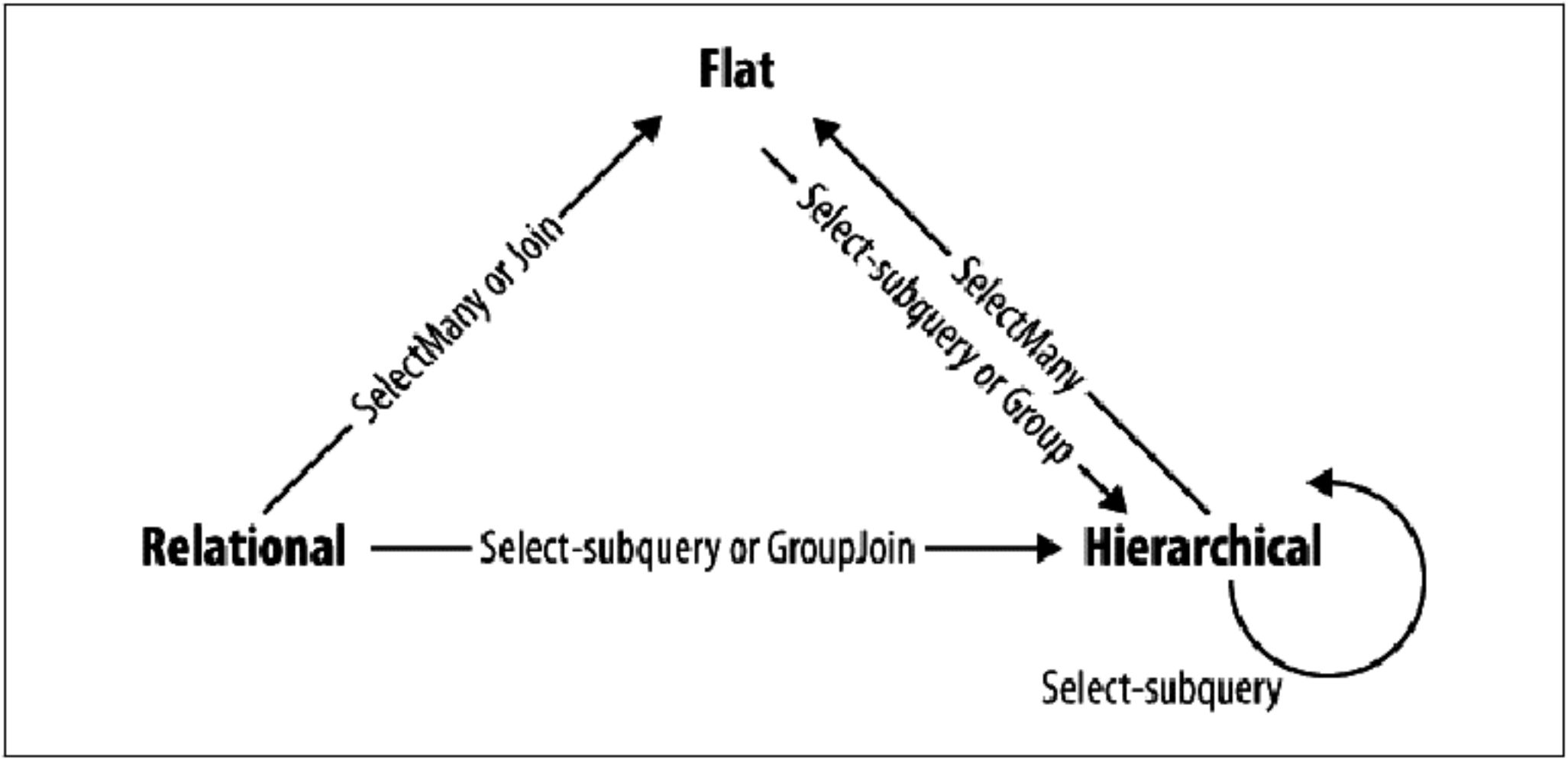


Figure 9-1. Shape-changing operators

Figure 9-1. Shape-changing operators

Filtering

`IEnumerable<TSource>→IEnumerable<TSource>`

Returns a subset of the original elements.

`Where, Take, TakeWhile, Skip, SkipWhile, Distinct`

Projecting

`IEnumerable<TSource>→IEnumerable<TResult>`

Transforms each element with a lambda function. `SelectMany` flattens nested sequences; `Select` and `SelectMany` perform inner joins, left outer joins, cross joins, and non-equi joins with LINQ to SQL and EF.

`Select, SelectMany`

Select, SelectMany

Joining

IEnumerable<TOuter> IEnumerable<TInner> ›IEnumerable<TResult>

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

Meshes elements of one sequence with another. The joining operators are designed to be efficient with local queries and support inner and left outer joins.

`Join, GroupJoin`

Ordering

`IEnumerable<TSource>→IOrderedEnumerable<TSource>`

Returns a reordering of a sequence.

`OrderBy, ThenBy, Reverse`

Grouping

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

Groups a sequence into subsequences.

Groups a sequence into subsequences.

GroupBy

Set operators

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

Takes two same-typed sequences and returns their commonality, sum, or difference.

Concat, Union, Intersect, Except

Zip operator

`IEnumerable<TFirst>, IEnumerable<TSecond>→IEnumerable<TResult>`

Enumerates two sequences in step, applying a function over each element pair.

Enumerates two sequences in step, applying a function over each element pair.

Conversion methods: Import

`IEnumerable→IEnumerable<TResult>`

`OfType, Cast`

Conversion methods: Export

`IEnumerable<TSource>`→An array, list, dictionary, lookup, or sequence

`ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable`

Sequence→Element or Scalar

The following query operators accept an input sequence and emit a single element or scalar value.

The following query operators accept an input sequence and emit a single element or scalar value.

Element operators

IEnumerable<TSource>→TSource

Picks a single element from a sequence.

`First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault, DefaultIfEmpty`

Aggregation methods

IEnumerable<TSource>→*scalar*

Performs a computation across a sequence, returning a scalar value (typically a

Performs a computation across a sequence, returning a scalar value (typically a number).

`Aggregate, Average, Count, LongCount, Sum, Max, Min`

Quantifiers

`IEnumerable<TSource>`→*bool*

An aggregation returning `true` or `false`.

`All, Any, Contains, SequenceEqual`

Void→Sequence

In the third and final category are query operators that produce an output sequence from scratch.

Generation methods

Generation methods

`void→IEnumerable<TResult>`

Manufactures a simple sequence.

`Empty, Range, Repeat`

Filtering

`IEnumerable<TSource>→ IEnumerable<TSource>`

Method	Description	SQL equivalents
`Where`	Returns a subset of elements that satisfy a given condition	`WHERE`

`Take`

`Skip`

`TakeWhile`

`SkipWhile`

TakeWhile

SkipWhile

Returns the first `count` elements and discards the rest

Ignores the first `count` elements and returns the rest

`WHERE ROW_NUMBER()...`

or `TOP` *n* subquery

`WHERE ROW_NUMBER()...`

Emits elements from the input sequence until the predicate is false

or `NOT IN (SELECT TOP` *n*`...)`

or `NOT IN (SELECT TOP` *n*`...)`

Exception thrown

Ignores elements from the input sequence until the predicate is false, and then emits the rest

Exception thrown

`Distinct`	Returns a sequence that excludes duplicates	`SELECT DISTINCT...`