# 3 The Essence of LINQ

NOW THAT YOU'VE SEEN several practical examples of LINQ's syntax, it is time to view the technology from a more theoretical perspective. This chapter covers the seven foundations on which an understanding of LINQ can be built. LINQ is

- Integrated
- Unitive
- Extensible
- Declarative
- Hierarchical
- Composable
- Transformative

These ideas may sound esoteric at first, but I believe you will find them quite easy to understand. LINQ has a fundamental simplicity and elegance. In this chapter and the next, we explore LINQ's architecture, giving you a chance to understand how it was built and why it was built that way. This chapter explains goals that LINQ aims to achieve. The next chapter explains each of the pieces of the LINQ architecture and shows how they come together to achieve those goals.

## Integrated

LINQ stands for Language Integrated Query. One of the central, and most important, features of LINQ is its integration of a flexible query syntax into the C# language.

Developers have many tools that have been crafted to neatly solve difficult tasks. Yet there are still dark corners in the development landscape. Querying data is one area in which developers frequently encounter problems with no clear resolution. LINQ aims to remove that uncertainty and to show a clearly defined path that is well-lit and easy to follow.

In Visual Studio 2005, attempts to query data in a SQL database from a C# program revealed an impedance mismatch between code and data. SQL is native to neither .NET nor C#. As a result, SQL code embedded in a C# program is neither type-checked nor IntelliSense-aware. From the perspective of a C# developer, SQL is shrouded in darkness.

Here is an example of one of several different techniques developers used in the past when querying data:

```
SqlConnection sqlConnection = new SqlConnection(connectString);
sqlConnection.Open();
System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;
sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader(CommandBehavior.CloseConnection)
```

Of these six lines of code, only the last two directly define a query. The rest of the lines involve setup code that allows developers to connect and call objects in the database. The query string shown in the next-to-last line is neither type-checked nor IntelliSense-aware.

After these six lines of code execute, the developers may have more work to do, because the data returned from the query is not readily addressable by an object-oriented programmer. You might have to write more lines of code to access this data, or convert it into a format that is easier to use.

The LINQ version of this same query is shorter, easier to read, color-coded, fully type-checked, and IntelliSense-aware. The result set is cleanly converted into a well-defined object-oriented format:

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");

var query = from c in db.Customers
            select c;
```

By fully integrating the syntax for querying data into .NET languages such as C# and VB, LINQ resolves a problem that has long plagued the development world. Queries become first-class citizens of our primary languages; they are both type-checked and supported by the powerful IntelliSense technology provided inside the Visual Studio IDE. LINQ brings the experience of writing queries into the well-lit world of the 21st century.

A few benefits accrue automatically as a result of integrating querying into the C# language:

- The syntax highlighting and IntelliSense support allow you to get more work done in less time. The Visual Studio editor automatically shows you the tables in your database, the correctly spelled names and types of your fields, and the operators you can use when querying data. This helps you save time and avoid careless mistakes.
- LINQ code is shorter and cleaner than traditional techniques for querying data and, therefore, is much easier to maintain.
- LINQ allows you to fully harness the power of your C# debugger while writing and maintaining queries. You can step through your queries and related code in your LINQ projects.

If language integration were the only feature that LINQ offered, that alone would have been a significant accomplishment. But we are only one-seventh of the way through our description of the foundations of LINQ. Many of the best and most important features are still to be covered.

## Unitive

Before LINQ, developers who queried data frequently needed to master multiple technologies. They needed to learn the following:

- SQL to query a database
- XPath, Dom, XSLT, or XQuery to query and transform XML data

- Web services to access some forms of remote data
- Looping and branching to query the collections in their own programs

These diverse APIs and technologies forced developers to frantically juggle their tight schedules while struggling to run similar queries against dissimilar data sources. Projects often encountered unexpected delays simply because it was easier to talk about querying XML, SQL, and other data than it was to actually implement the queries against these diverse data sources. If you have to juggle too many technologies, eventually something important will break.

LINQ simplifies these tasks by providing a single, unified method for querying diverse types of data. Developers don't have to master a new technology simply because they want to query a new data source. They can call on their knowledge of querying local collections when they query relational data, and vice versa.

This point was illustrated in the preceding chapter, where you saw three very similar queries that drew data from three different data sources: objects, an SQL database, and XML:

```
var query = from c in GetCustomers()
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from c in db.Customers
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from x in customers.Descendants("Customer")
            where x.Attribute("City").Value == "Mexico D.F."
            select x;
```

As you can see, the syntax for each of these queries is not identical, but it is very similar. This illustrates one of LINQ's core strengths: a single, unitive syntax can be used to query diverse types of data. It is not that you never have to scale a learning curve when approaching a new data source, but only that the principles, overall syntax, and theory are the same even if some of the details differ.

You enjoy two primary benefits because LINQ is unitive:

- The similar syntax used in all LINQ queries helps you quickly get up to speed when querying new data sources.
- Your code is easier to maintain, because you are using the same syntax regardless of the type of data you query.

Although it arises naturally from this discussion, it is worth noting that SQL and other query languages do not have this capability to access multiple data sources with a single syntax. Those who advocate using SQL or the DOM instead of LINQ often forget that their decision forces their team to invest additional time in learning these diverse technologies.

## Extensible Provider Model

In this text I have tended to define LINQ as a tool for querying SQL, XML, and the collections in a program. Strictly speaking, this is not an accurate description of LINQ. Although such a view is useful when you first encounter LINQ, it needs to be abandoned if you want to gain deeper insight. LINQ is not designed to query any particular data source; rather, it is a technology for defining *providers* that can be used to access any arbitrary data source. LINQ happens to ship with providers for querying SQL, XML, and objects, but this was simply a practical decision, not a preordained necessity.

LINQ provides developers with a syntax for querying data. This syntax is enabled by a series of C# 3.0 and C# 2.0 features. These include lambdas, iterator blocks, expression trees, anonymous types, type inference, query expressions, and extension methods. All of these features are covered in this book. For now you need only understand that they make LINQ possible.

When Visual Studio 2008 shipped, Microsoft employees frequently showed the image shown in Figure 3.1. Although people tend to think of LINQ as a means of enabling access to these data sources, this diagram actually depicts nothing more than the set of LINQ providers that were implemented by Microsoft at the time Visual Studio shipped. Granted, the team carefully planned which providers they wanted to ship, but their decisions were based on strategic, rather than technical, criteria.

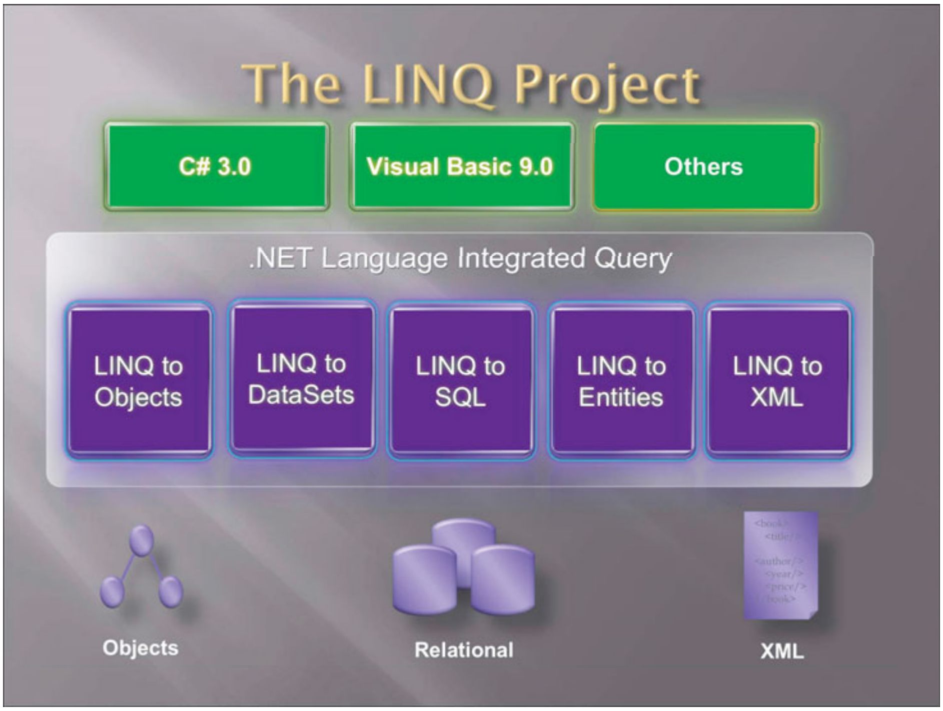


**FIGURE 3.1** VB and C# ship with LINQ providers for databases, XML, and data structures found in a typical program.

Using the LINQ provider model, developers can extend LINQ to query other data sources besides those shown in Figure 3.1. The following are a few of the data sources currently enabled by third-party LINQ providers:

| | |
|---|---|
| LINQ Extender | LINQ to Google |
| LINQ over C# project | LINQ to Indexes |
| LINQ to Active Directory | LINQ to `IQueryable` |
| LINQ to Amazon | LINQ to JavaScript |
| LINQ to Bindable Sources | LINQ to JSON |
| LINQ to CRM | LINQ to LDAP |
| LINQ to Excel | LINQ to LLBLGen Pro |
| LINQ to Expressions | LINQ to Lucene |
| LINQ to Flickr | LINQ to Metaweb |
| LINQ to Geo | LINQ to MySQL |

| | |
|---|---|
| LINQ to NCover | LINQ to Sharepoint |
| LINQ to NHibernate | LINQ to SimpleDB |
| LINQ to Opf3 | LINQ to Streams |
| LINQ to Parallel (PLINQ) | LINQ to WebQueries |
| LINQ to RDF Files | LINQ to WMI |

These projects are of varying quality. Some, such as the LINQ Extender and LINQ to `IQueryable`, are merely tools for helping developers create providers. Nevertheless, you can see that an active community is interested in creating LINQ providers, and this community is producing some interesting products. By the time you read this, I'm sure the list of providers will be longer. See Appendix A for information on how to get updated information on existing providers.

One easily available provider called LinqToTerraServer can be found among the downloadable samples that ship with Visual Studio 2008. You can download the VS samples from the release tab found at http://code.msdn.microsoft.com/csharpsamples.

After unzipping the download, if you look in the ...\LinqSamples\WebServiceLinqProvider directory, you will find a sample called LinqToTerraServer. The TerraServer web site, http://terraserver-usa.com, is a vast repository of pictures and information about geographic information. The LinqToTerraServer example shows you how to create a LINQ provider that queries the web services provided on the TerraServer site. For example, the following query returns all U.S. cities and towns named Portland:

```
var query1 = from place in terraPlaces
             where place.Name == "Portland"
             select new { place.Name, place.State };
```

This query returns a number of locations, but here are a few of the more prominent:

```
{ Name = Portland, State = Indiana }
{ Name = Portland, State = Maine }
{ Name = Portland, State = Michigan }
{ Name = Portland, State = Oregon }
{ Name = Portland, State = Texas }
{ Name = Portland, State = Alabama }
{ Name = Portland, State = Arkansas }
{ Name = Portland, State = Colorado }
```

In Chapter 17, "LINQ Everywhere," you will see examples of several other providers, including LINQ to Flickr and LINQ to SharePoint. It is not easy to create a provider.. After the code is written, however, it is easy to use the provider. In fact, you should already have enough familiarity with LINQ to see that it would be easy to modify the preceding query to suit your own purposes.

The LINQ provider model has hidden benefits that might not be evident at first glance:

- It is relatively open to examination and modification. As you read the next few chapters, you will find that most of the LINQ query pipeline is accessible to developers.
- It allows developers to be intelligent about how queries execute. You can get a surprising degree of control over the execution of a query. If you care about optimizing a query, in many cases you can optimize it, because you can see how it works.
- You can create a provider to publicize a data source that you have created. For instance, if you have a web service that you want C# developers to access, you can create a provider to give them a simple, extensible way to access your data.

I will return to the subject of LINQ providers later in the book. In this chapter, my goal is simply to make it clear that LINQ is extensible, and that its provider model is the basis on which each LINQ query model is built.

## Query Operators

You don't always need to use a LINQ provider to run queries against what might—at least at first—appear to be nontraditional data sources. By using the LINQ to Objects provider, and a set of built-in LINQ operators, you can run queries against a data source that does not look at all like XML or SQL data. For instance, LINQ to Objects gives you access to the reflection model that is built into C#.

The following query retrieves all the methods of the `string` class that are static:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            select m;
```

The following are a few of the many results that this query returns:

```
System.String Join(System.String, System.String[])
System.String Join(System.String, System.String[], Int32, Int32)
Boolean Equals(System.String, System.String)
Boolean Equals(System.String, System.String, System.StringComparison)
Boolean op_Equality(System.String, System.String)
Boolean op_Inequality(System.String, System.String)
Boolean IsNullOrEmpty(System.String)
Int32 Compare(System.String, System.String)
Int32 Compare(System.String, System.String, Boolean)
Int32 Compare(System.String, System.String, System.StringComparison)
```

Using the power of LINQ, it is easy to drill into these methods to find out more about them. In particular, LINQ uses the extension methods mentioned in the preceding section to define a set of methods that can perform specific query operations such as ordering and grouping data. For instance, the following query retrieves the methods of the `string` class that are static, finds out how many overloads each method has, and then orders them first by the number of overloads and then alphabetically:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            orderby m.Name
            group m by m.Name into g
            orderby g.Count()
            select new { Name = g.Key, Overloads = g.Count() };

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

The results of this query look like this:

```
{ Overloads = 1, Name = Copy }
{ Overloads = 1, Name = Intern }
{ Overloads = 1, Name = IsInterned }
{ Overloads = 1, Name = IsNullOrEmpty }
{ Overloads = 1, Name = op_Equality }
{ Overloads = 1, Name = op_Inequality }
{ Overloads = 2, Name = CompareOrdinal }
{ Overloads = 2, Name = Equals }
{ Overloads = 2, Name = Join }
{ Overloads = 5, Name = Format }
{ Overloads = 9, Name = Concat }
{ Overloads = 10, Name = Compare }
```

This makes it obvious that `Format`, `Compare`, and `Concat` are the most frequently overloaded methods of the `string` class, and it presents all the methods with the same number of overloads in alphabetical order.

You can run this code in your own copy of Visual Studio because the LINQ to Objects provider ships with C# 3.0. Other third-party extensions to LINQ, such as LINQ to Amazon, are not included with Visual Studio. If you want to run a sample based on LINQ to Amazon or some other provider that does not ship with Visual Studio, you must download and install the provider before you can use it.

## Declarative: Not How, But What

LINQ is declarative, not imperative. It allows developers to simply state what they want to do without worrying about how it is done.

Imperative programming requires developers to define step by step how code should be executed. To give directions in an imperative fashion, you say, "Go to 1st Street, turn left onto Main, drive two blocks, turn right onto Maple, and stop at the third house on the left." The declarative version might sound something like this: "Drive to Sue's house." One says *how* to do something; the other says *what* needs to be done.

The declarative style has two advantages over the imperative style:

- It does not force the traveler to memorize a long set of instructions.
- It allows the traveler to optimize the route when possible.

It should be obvious that there is little opportunity to optimize the first set of instructions for getting to Sue's house: You simply have to follow them by rote. The second set, however, allows the traveler to use his or her knowledge of the neighborhood to find a shortcut. For instance, a bike might be the best way to travel at rush hour, whereas a car might be best at night. On occasion, going on foot and cutting through the local park might be the best solution.

Here is another example of the difference between declarative and imperative code:

```
// imperative style
List<int> imperativeList = new List<int>();
imperativeList.Add(1);
imperativeList.Add(2);
imperativeList.Add(3);

// declarative style
List<int> declaractiveList = new List<int> { 1, 2, 3 };
```

The first example details exactly how to add items to a list. The second example states what you want to do and allows the compiler to figure out the best way to do it. As you will learn in the next chapter, both styles are valid C# 3.0 syntax. The declarative form of this code, however, is shorter, easier to understand, easier to maintain, and, at least in theory, leaves the compiler free to optimize how a task is performed.

These two styles differ in both the amount of detail they require a developer to master and the amount of freedom that each affords the compiler. Detailed instructions not only place a burden on the developer, but also restrict the compiler's capability to optimize code.

Let's consider another example of the imperative style of programming. As developers, we frequently end up in a situation where we are dealing with a list of lists:

```
List<int> list01 = new List<int> { 1, 2, 3 };
List<int> list02 = new List<int> { 4, 5, 6 };
List<int> list03 = new List<int> { 7, 8, 9 };

List<List<int>> lists = new List<List<int>> { list01, list02, list03 };
```

Here is imperative code for accessing the members of this list:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
```

```
        {
            newList.Add(number);
        }
    }
```

This code produces a single list containing all the data from the three nested lists:

```
1
2
3
4
5
6
7
8
9
```

Notice that we have to write nested `for` loops to allow access to our data. In a simple case like this, nested loops are not terribly complicated to use, but they can become very cumbersome in more complex problem domains.

Contrast this code with the declarative style used in a LINQ program:

```
var newList = from list in lists
              from num in list
              select num;
```

You can access the results of these two "query techniques" in the same way:

```
foreach (var item in newList)
{
    Console.WriteLine(item);
}
```

This code writes the results of either query, producing identical results, regardless of whether you used the imperative or declarative technique to query the data:

```
1
2
3
4
5
6
7
8
9
```

The difference here is not in the query's results, or in how we access the results, but in how we compose our query against our nested list. The imperative style can sometimes be verbose and hard to read. The declarative code is usually short and easy to read and scales more easily to complex cases. For instance, you can add an `orderby` clause to reverse the order of the integers in your result set:

```
var query = from list in lists
            from num in list
            orderby num descending
            select num;
```

You probably know how to achieve the same results using the imperative style. But it was knowledge that you had to struggle to learn, and it is knowledge that applies only to working with sequences of numbers stored in a `List<T>`. The LINQ code for reordering results, however, is easy to understand. It can be used to reorder not only nested collections, but also SQL data, XML data, or the many other data sources we query using LINQ.

To get the even numbers from our nested lists, we need only do this:

```
var query = from list in lists
            from num in list
            where num % 2 == 0
            orderby num descending
            select num;
```

Contrast this code with the imperative equivalent:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
    {
        if (number % 2 == 0)
        {
            newList.Add(number);
        }
    }
}

newList.Reverse();
```

This imperative style of programming now has an `if` block nested inside the nested `foreach` loops. This is not only verbose and applicable to only a specific type of data, it also can be like a straight jacket for both the compiler and the developer. Commands must be issued and followed in a rote fashion, leaving little room for optimizations.

The equivalent LINQ query expression does not describe in a step-by-step fashion how to query our list of lists. It simply lets the developer state what he wants to do and lets the compiler determine the best path to the destination.

After nearly 50 years of steady development, the possibilities inherent in imperative programming have been extensively explored. Innovations in the field are now rare. Declarative programming, on the other hand, offers opportunities for growth. Although it is not a new field of study, it is still rich in possibilities.

## Use the Right Tool for the Job

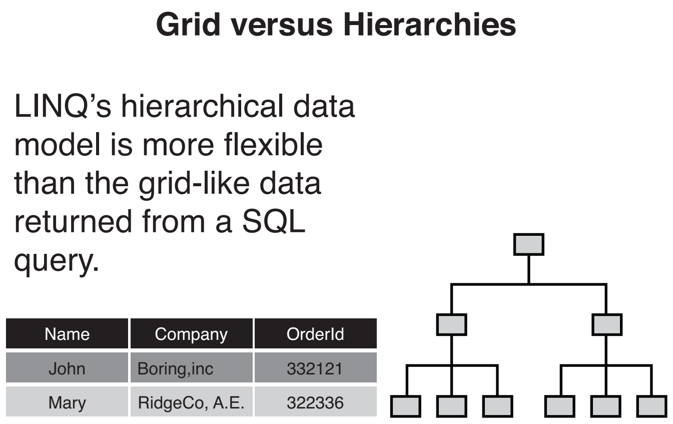
In extolling the virtues of LINQ's declarative syntax, I should be careful not to overstate my case. For instance, the LINQ operator called `ToList` is provided to allow developers to easily translate the sequence of results returned by a LINQ query into a traditional `List<T>`. This functionality is useful because some operations, such as randomly accessing items in a list (`myList[2]`), are more easily performed using the imperative syntax. One of the great virtues of C# 3.0 is that it allows you to easily move between imperative and declarative syntax, allowing you to choose the best tool for the job. My job right now is to help you understand the value of LINQ and the declarative style of programming. LINQ is indeed a very powerful and useful tool, but it is not the solution to all your problems.

Because LINQ is a new technology from Microsoft, you might find it a bit jarring to see me write that declarative programming is not new. In fact, declarative code has been with us nearly as long as imperative code. Some older languages such as LISP (which was first specified in 1958) make heavy use of the declarative style of programming. Haskel and F# are examples of other languages that use it extensively. One reason LINQ and SQL look so much alike is that they are both forms of declarative programming.

The point of LINQ is not that it will replace SQL, but that it will bring the benefits of SQL to C# developers. LINQ is a technology for enabling a SQL-like declarative programming style inside a native C# program. It brings you the benefits of SQL but adds declarative syntax, as well as syntax highlighting, IntelliSense support, type checking, debugging support, the ability to query multiple data sources with the same syntax, and much more.

## Hierarchical

Complex relationships can be expressed in a relational database, but the results of a SQL query can take only one shape: a rectangular grid. LINQ has no such restrictions. Built into its very foundation is the idea that data is hierarchical (see Figure 3.2). If you want to, you can write LINQ queries that return flat, SQL-like datasets, but this is an option, not a necessity.



LINQ's hierarchical data model is more flexible than the grid-like data returned from a SQL query.

| Name | Company | OrderId |
|---|---|---|
| John | Boring,inc | 332121 |
| Mary | RidgeCo, A.E. | 322336 |

FIGURE 3.2 Both object-oriented languages and the developers who use them have a natural tendency to think in terms of hierarchies. SQL data is arranged in a simple grid.

Consider a simple relational database that has tables called Customers, Orders, and OrderDetails. It is possible to capture the relationship between these tables in a SQL database, but you cannot directly depict the relationship in the results of a single query. Instead, you are forced to show the result as a join that binds the tables into a single array of columns and rows.

LINQ, on the other hand, can return a set of `Customer` objects, each of which owns a set of 0-to-*n* `Orders`. Each `Order` can be associated with a set of `OrderDetails`. This is a classic hierarchical relationship that can be perfectly expressed with a set of objects:

```
Customer
    Orders
        OrderDetails
```

Consider the following simple hierarchical query that captures the relationship between two objects:

```
var query = from c in db.Customers
            select new { City = c.City,
                         orders = from o in c.Orders
                         select new { o.OrderID }
                       };
```

This query asks for the city in which a customer lives and a list of the orders the person has made. Rather than returning a rectangular dataset as a SQL query would, this query returns hierarchical data that lists the city associated with each customer and the ID associated with each order:

```
City=Helsinki   orders=...
  orders: OrderID=10615
  orders: OrderID=10673
  orders: OrderID=10695
  orders: OrderID=10873
  orders: OrderID=10879
  orders: OrderID=10910
  orders: OrderID=11005
City=Warszawa   orders=...
  orders: OrderID=10374
  orders: OrderID=10611
  orders: OrderID=10792
  orders: OrderID=10870
  orders: OrderID=10906
  orders: OrderID=10998
```

This result set is multidimensional, nesting one set of columns and rows inside another set of columns and rows.

Look again at the query, and notice how we gain access to the Orders table:

```
orders = from o in c.Orders
```

The identifier `c` is an instance of a `Customer` object. As you will learn later in the book, LINQ to SQL has tools for automatically generating `Customer` objects given the presence of the Customer table in the database. Here you can see that the `Customer` object is not flat; instead, it contains a set of nested `Order` objects.

Listing 3.1 shows a simplified version of the `Customer` object that is automatically generated by the LINQ to SQL designer. Notice how LINQ to SQL wraps the fields of the Customer table. Later in this book, you will learn how to automatically generate `Customer` objects that wrap the fields of a Customer table.

**LISTING 3.1 A Simplified Version of the `Customer` Object That the LINQ to SQL Designer Generates Automatically**

```
public partial class Customer
{
     ... // Code omitted here
     private string _CustomerID;
     private string _CompanyName;
     private string _ContactName;
     private string _ContactTitle;
     private string _Address;
     private string _City;
     private string _Region;
     private string _PostalCode;
     private string _Country;
     private string _Phone;
     private string _Fax;
     private EntitySet<Order> _Orders;
     ... // Code omitted here
}
```

The first 11 private fields of the `Customer` object simply reference the fields of the Customer table in the database. Taken together, they provide a location to store the data from a single row of the Customer table. Notice, however, the last item, which is a collection of `Order` objects. Because it is bound to the Orders table in a one-to-many relationship, each customer has from 0-to-*n* orders associated with it, and LINQ to SQL stores those orders in this field. This automatically gives you a hierarchical view of your data.

The same thing is true of the Order table, only it shows not a one-to-many relationship with the Customer table, but a one-to-one relationship:

```
public partial class Order
{
    ... // Code omitted here
    private int _OrderID;
    private string _CustomerID;
    private System.Nullable<int> _EmployeeID;
    private System.Nullable<System.DateTime> _OrderDate;
    private System.Nullable<System.DateTime> _RequiredDate;
    private System.Nullable<System.DateTime> _ShippedDate;
    private System.Nullable<int> _ShipVia;
    private System.Nullable<decimal> _Freight;
    private string _ShipName;
    private string _ShipAddress;
    private string _ShipCity;
    private string _ShipRegion;
    private string _ShipPostalCode;
    private string _ShipCountry;
    private EntityRef<Customer> _Customer;
    ... // Code omitted here
}
```

Again we see all the fields of the Orders table, their types, and whether they can be set to Null. The difference here is that the last field points back to the Customer table not with an `EntitySet<T>`, but an `EntityRef<T>`. This is not the proper place to delve into the `EntitySet` and `EntityRef` classes. However, it should be obvious to you that an `EntitySet` refers to a set of objects, and an `EntityRef` references a single object. Thus, an `EntitySet` captures a one-to-many relationship, and an `EntityRef` captures a one-to-one relationship.

The point to take away from this discussion is that LINQ to SQL captures not a flat view of your data, but a hierarchical view. A `Customer` class is connected to a set of orders in a clearly defined hierarchical relationship, and each order is related to the customer who owns it. LINQ gives you a hierarchical view of your data.

In a simple case like this, such a hierarchical relationship has obvious utility, but it is possible to imagine getting along without it. More complex queries, however, are obviously greatly simplified by this architecture. Consider the following LINQ to SQL query:

```
var query = from c in db.Customers
            where c.CompanyName == companyName
            from o in c.Orders
            from x in o.Order_Details
            where x.Product.Category.CategoryName == "Confections"
            orderby x.Product.ProductName
            group x by x.Product.ProductName into g
            orderby g.Count()
            select new { Count = g.Count(), Product = g.Key };
```

Here we use LINQ's hierarchical structure to move from the Customers table to the Orders table to the Order_Details table without breaking a sweat:

```
var query = from c in db.Customers
            from o in c.Orders
            from x in o.Order_Details
```

The next line really helps show the power of LINQ hierarchies:

```
where x.Product.Category.CategoryName == "Confections"
```

The identifier x represents an instance of a class containing the data from a row of the Order_Details table. Order_Details has a relationship with the Product table, which has a relationship with the Category table, which has a field called `CategoryName`. We can slice right through that complex relationship by simply writing this:

```
x.Product.Category.CategoryName
```

LINQ's hierarchical structure shines a clarifying light on the relational data in your programs. Even complex relational models become intuitive and easy to manipulate.

We can then order and group the results of our query with a few simple LINQ operators:

```
orderby x.Product.ProductName
group x by x.Product.ProductName into g
orderby g.Count()
```

Trying to write the equivalent code using a more conventional C# style of programming is an exercise that might take two or three pages of convoluted code and involve a number of nested loops and `if` statements. Even writing the same query in standard SQL would be a challenge for many developers. Here we perform the whole operation in nine easy-to-read lines of code.

In this section, I have introduced you to the power of LINQ's hierarchical style of programming without delving into the details of how such queries work. Later in this book you will learn how easy it is to compose your own hierarchical queries. For now you only need to understand two simple points:

- There is a big difference between LINQ's hierarchical structure and the flat, rectangular columns and rows returned by an SQL query.
- Many benefits arise from this more powerful structure. These include the intuitive structure of the data and the ease with which you can write queries against this model.

## Composable

The last two foundations of LINQ shed light on its flexibility and power. If you understand these two features and how to use them, you will be able to tap into some very powerful technology. Of course, this chapter only introduces these features; they are discussed in more detail in the rest of the book.

LINQ queries are composable: You can combine them in multiple ways, and one query can be used as the building block for yet another query. To see how this works, let's look at a simple query:

```
var query = from customer in db.Customers
            where customer.City == "Paris"
            select customer;
```

The variable that is returned from the query is sometimes called a computation. If you write a `foreach` loop and display the address field from the customers returned by this computation, you see the following output:

```
265, boulevard Charonne
25, rue Lauriston
```

You can now write a second query against the results of this query:

```
query2 = from customer in query
        where customer.Address.StartsWith("25")
        select customer;
```

Notice that the last word in the first line of this query is the computation returned from the previous query. This second query produces the following output:

```
25, rue Lauriston
```

LINQ to Objects queries are composable because they operate on and usually return variables of type `IEnumerable<T>`. In other words, LINQ queries typically follow this pattern:

```
IEnumerable<T> query = from x in IEnumerable<T>
                       select x;
```

This is a simple mechanism to understand, but it yields powerful results. It allows you to take complex problems, break them into manageable pieces, and solve them with code that is easy to understand and easy to maintain. You will hear much more about `IEnumerable<T>` in the next chapter.

The next chapter also details a feature called deferred execution. Although it can be confusing to newcomers, one of the benefits of deferred execution is that it allows you to compose multiple queries and string them together without necessarily needing to have each query entail an expensive hit against the server. Instead, three or four queries can "execute" without ever sending a query across the wire to your database. Then, when you need to access the result from your query, a SQL statement is written that combines the results of all your queries and sends it across the wire only once. Deferred execution is a powerful feature, but you need to wait until the next chapter for a full explanation of how and why it works. The key point to grasp now is that it enables you to compose multiple queries as shown here, without having to take an expensive hit each time one "executes."

### Discreet Computations and PLINQ

LINQ queries are not only composable, but also discreet. In other words, the computation returned by a query is a single self-contained expression with only a single entry point. This has important consequences for a field of study called Parallel LINQ (PLINQ). Because each computation returned by a query is discreet, it can easily be run concurrently on its own thread. PLINQ is discussed briefly in Chapter 17, “LINQ Everywhere.

## Transformative

SQL is poor at transformations, so we are unaccustomed to thinking about query languages as a tool for converting data from one format to another. Instead, we usually use specialized tools such as XSLT or brute-force techniques to transform data.

LINQ, however, has transformational powers built directly into its syntax. We can compose a LINQ query against a SQL database that effortlessly performs a variety of transforms. For instance, with LINQ it is easy to transform the result of a SQL query into a hierarchical XML document. You can also easily transform one XML document into another with a different structure. SQL data is transformed into a hierarchical set of objects automatically when you use LINQ to SQL. In short, LINQ is very good at transforming data, and this adds a new dimension to our conception of what we can do with a query language.

Listing 3.2 shows code that takes the results of a query against relational data and transforms it into XML.

**Listing 3.2 A Simple Query That Transforms the Results of a LINQ to SQL Query into XML**

```
var query = new XElement("Orders", from c in db.Customers
              where c.City == "Paris"
              select new XElement("Order",
                 new XAttribute("Address", c.Address),
                 new XAttribute(“City”, c.City)));
```

Embedded in this query is a simple LINQ to SQL query that returns the `Address` and `City` fields from all the customers who live in Paris. In Listing 3.3 I’ve stripped away the LINQ to XML code from Listing 3.2 to show you the underlying LINQ to SQL query.

**Listing 3.3 The Simple LINQ to SQL Query Found at the Heart of Listing 3.2**

```
var query = from c in db.Customers
            where c.City == "Paris"
            select new { c.Address, c.City };
```

Here is the output from Listing 3.3:

```
265, boulevard Charonne
25, rue Lauriston
```

Here is the output from Listing 3.2:

```
<Orders>
  <Order Address="265, boulevard Charonne" City="Paris" />
  <Order Address="25, rue Lauriston" City="Paris" />
</Orders>
```

As you can see, the code in Listing 3.2 performs a transform on the results of the LINQ to SQL query, converting it into XML data.

Because LINQ is composable, the following query could then be used to run a second transform on this data:

```
var query1 = new XElement("Orders", new XAttribute("City", "Paris"),
    from x in query.Descendants("Order")
    where x.Attribute("City").Value == "Paris"
    select new XElement("Address", x.Attribute("Address").Value));
```

This query takes the XML results of the first query and transforms that XML into the following format:

```
<Orders City="Paris">
  <Address>265, boulevard Charonne</Address>
  <Address>25, rue Lauriston</Address>
</Orders>
```

LINQ is constantly transforming one type of data into another type. It takes relational data and transforms it into objects; it takes XML and transforms it into relational data. Because LINQ is extensible, it is at least theoretically possible to use it to tear down the walls that separate any two arbitrary data domains.

Because LINQ is both composable and transformative, you can use it in a number of unexpected ways:

- You can compose multiple queries, linking them in discrete chunks. This often allows you to write code that is easier to understand and maintain than traditional nested SQL queries.
- You can easily transform data from one data source into some other type. For instance, you can transform SQL data into XML.
- Even if you do not switch data sources, you can still transform the shape of data. For instance, you can transform one XML format into another format. If you look back at the section "Declarative: Not How, But What," you will see that we transformed data that was stored in nested lists into data that was stored in a single list. These kinds of transformations are easy with LINQ.

## Summary

In this chapter you have read about the foundations of LINQ. These foundations represent the core architectural ideas on which LINQ is built. Taken together, they form the essence of LINQ. We can summarize these foundations by saying the following about LINQ:

- It is a technique for querying data that is *integrated* into .NET languages such as C# and VB. As such, it is both strongly typed and IntelliSense-aware.
- It has a single *unitive* syntax for querying multiple data sources such as relational data and XML data.
- It is *extensible*; talented developers can write providers that allow LINQ to query any arbitrary data source.
- It uses a *declarative* syntax that allows developers to tell the compiler or provider what to do, not how to do it.
- It is *hierarchical*, in that it provides a rich, object-oriented view of data.
- It is *composable*, in that the results of one query can be used by a second query, and one query can be a subclause of another query. In many cases, this can be done without forcing the execution of any one query until the developer wants that execution to take place.

- It is *transformative*, in that the results of a LINQ query against one data source can be morphed into a second format. For instance, a query against a SQL database can produce an XML file as output.

Scattered throughout this chapter are references to some of the important benefits of LINQ that emerge from these building blocks. Although these benefits were mentioned throughout this chapter, I'll bring them together here in one place as a way of reviewing and summarizing the material discussed in this chapter:

- Because LINQ is integrated into the C# language, it provides syntax highlighting and IntelliSense. These features make it easy to write accurate queries and to discover mistakes at design time.
- Because LINQ queries are integrated into the C# language, it is possible for you to write code much faster than if you were writing old-style queries. In some cases, developers have seen their development time cut in half.
- The integration of queries into the C# language also makes it easy for you to step through your queries with the integrated debugger.
- The hierarchical feature of LINQ allows you to easily see the relationship between tables, thereby making it easy to quickly compose queries that join multiple tables.
- The unitive foundation of LINQ allows you to use a single LINQ syntax when querying multiple data sources. This allows you to get up to speed on new technologies much more quickly. If you know how to use LINQ to Objects, it is not hard to learn how to use LINQ to SQL, and it is relatively easy to master LINQ to XML.
- Because LINQ is extensible, you can use your knowledge of LINQ to make new types of data sources queriable.
- After creating or discovering a new LINQ provider, you can leverage your knowledge of LINQ to quickly understand how to write queries against these new data sources.
- Because LINQ is composable, you can easily join multiple data sources in a single query, or in a series of related queries.

- The composable feature of LINQ also makes it easy to break complex problems into a series of short, comprehensible queries that are easy to debug.
- The transformational features of LINQ make it easy to convert data of one type into a second type. For instance, you can easily transform SQL data into XML data using LINQ.
- Because LINQ is declarative, it usually allows you to write concise code that is easy to understand and maintain.
- The compiler and provider translate declarative code into the code that is actually executed. As a rule, LINQ knows more than the average developer about how to write highly optimized, efficient code. For instance, the provider might optimize or reduce nested queries.
- LINQ is a transparent process, not a black box. If you are concerned about how a particular query executes, you usually have a way to examine what is taking place and to introduce optimizations into your query.

This chapter touched on many other benefits of LINQ. These are described throughout this book. This entire text is designed to make you aware of the benefits that LINQ can bring to your development process. It also shows you how to write code that makes those benefits available to you and the other developers on your team.

The more you understand LINQ, the more useful it will be to you. As I have dug more deeply into this technology, I have found myself integrating LINQ into many different parts of my development process. When I use LINQ, I can get more work done in less time. The more I use it, the more completely these benefits accrue.