



LINQ

in Action

BONUS CHAPTER

Fabrice Marguerie
Steve Eichert
Jim Wooley

FOREWORD BY Matt Warren
Principal Architect, Microsoft

 MANNING

contents

foreword xv
preface xvii
acknowledgments xix
about this book xxii

PART 1 GETTING STARTED..... 1

1 *Introducing LINQ* 3

1.1 What is LINQ? 4

Overview 5 ■ *LINQ as a toolset* 6 ■ *LINQ as language extensions* 7

1.2 Why do we need LINQ? 9

Common problems 10 ■ *Addressing a paradigm mismatch* 12
LINQ to the rescue 18

1.3 Design goals and origins of LINQ 19

The goals of the LINQ project 20 ■ *A bit of history* 21

1.4 First steps with LINQ to Objects: Querying collections in memory 23

What you need to get started 23 ■ *Hello LINQ to Objects* 25

- 1.5 First steps with LINQ to XML: Querying XML documents 29
 - Why we need LINQ to XML* 30 ■ *Hello LINQ to XML* 32
- 1.6 First steps with LINQ to SQL: Querying relational databases 37
 - Overview of LINQ to SQL's features* 37 ■ *Hello LINQ to SQL* 38 ■ *A closer look at LINQ to SQL* 42
- 1.7 Summary 42

2 C# and VB.NET language enhancements 44

- 2.1 Discovering the new language enhancements 45
 - Generating a list of running processes* 46 ■ *Grouping results into a class* 47
- 2.2 Implicitly typed local variables 49
 - Syntax* 49 ■ *Improving our example using implicitly typed local variables* 50
- 2.3 Object and collection initializers 52
 - The need for object initializers* 52 ■ *Collection initializers* 53
 - Improving our example using an object initializer* 54
- 2.4 Lambda expressions 55
 - A refresher on delegates* 56 ■ *Anonymous methods* 58 ■ *Introducing lambda expressions* 58
- 2.5 Extension methods 64
 - Creating a sample extension method* 64 ■ *More examples using LINQ's standard query operators* 68
 - Extension methods in action in our example* 70
 - Warnings* 71
- 2.6 Anonymous types 73
 - Using anonymous types to group data into an object* 74
 - Types without names, but types nonetheless* 74
 - Improving our example using anonymous types* 76 ■ *Limitations* 76
- 2.7 Summary 79

3 LINQ building blocks 82

- 3.1 How LINQ extends .NET 83
 - Refresher on the language extensions* 83 ■ *The key elements of the LINQ foundation* 85
- 3.2 Introducing sequences 85
 - IEnumerable<T>* 86 ■ *Refresher on iterators* 87
 - Deferred query execution* 89
- 3.3 Introducing query operators 93
 - What makes a query operator?* 93 ■ *The standard query operators* 96
- 3.4 Introducing query expressions 97
 - What is a query expression?* 98 ■ *Writing query expressions* 98 ■ *How the standard query operators relate to query expressions* 100 ■ *Limitations* 102
- 3.5 Introducing expression trees 104
 - Return of the lambda expressions* 105 ■ *What are expression trees?* 105 ■ *IQueryable, deferred query execution redux* 108
- 3.6 LINQ DLLs and namespaces 109
- 3.7 Summary 111

PART 2 QUERYING OBJECTS IN MEMORY..... 113

4 Getting familiar with LINQ to Objects 115

- 4.1 Introducing our running example 116
 - Goals* 116 ■ *Features* 117 ■ *The business entities* 117
 - Database schema* 118 ■ *Sample data* 118
- 4.2 Using LINQ with in-memory collections 121
 - What can we query?* 121 ■ *Supported operations* 126
- 4.3 Using LINQ with ASP.NET and Windows Forms 126
 - Data binding for web applications* 127 ■ *Data binding for Windows Forms applications* 133

- 4.4 Focus on major standard query operators 137
 - Where, the restriction operator* 138 ■ *Using projection operators* 139 ■ *Using Distinct* 142 ■ *Using conversion operators* 143 ■ *Using aggregate operators* 145
- 4.5 Creating views on an object graph in memory 146
 - Sorting* 146 ■ *Nested queries* 147 ■ *Grouping* 150
 - Using joins* 151 ■ *Partitioning* 155
- 4.6 Summary 159

5 *Beyond basic in-memory queries* 160

- 5.1 Common scenarios 161
 - Querying nongeneric collections* 162 ■ *Grouping by multiple criteria* 164 ■ *Dynamic queries* 167 ■ *LINQ to Text Files* 178
- 5.2 Design patterns 180
 - The Functional Construction pattern* 181 ■ *The ForEach pattern* 184
- 5.3 Performance considerations 186
 - Favor a streaming approach* 187 ■ *Be careful about immediate execution* 189 ■ *Will LINQ to Objects hurt the performance of my code?* 191 ■ *Getting an idea about the overhead of LINQ to Objects* 195 ■ *Performance versus conciseness: A cruel dilemma?* 198
- 5.4 Summary 200

PART 3 *QUERYING RELATIONAL DATA*..... 203

6 *Getting started with LINQ to SQL* 205

- 6.1 Jump into LINQ to SQL 207
 - Setting up the object mapping* 209 ■ *Setting up the DataContext* 212
- 6.2 Reading data with LINQ to SQL 212
- 6.3 Refining our queries 217
 - Filtering* 217 ■ *Sorting and grouping* 219
 - Aggregation* 221 ■ *Joining* 222

- 6.4 Working with object trees 226
- 6.5 When is my data loaded and why does it matter? 229
 - Lazy loading* 229 ■ *Loading details immediately* 231
- 6.6 Updating data 233
- 6.7 Summary 236

7 *Peeking under the covers of LINQ to SQL* 237

- 7.1 Mapping objects to relational data 238
 - Using inline attributes* 239 ■ *Mapping with external XML files* 245 ■ *Using the SqlMetal tool* 247 ■ *The LINQ to SQL Designer* 249
- 7.2 Translating query expressions to SQL 252
 - IQueryable* 252 ■ *Expression trees* 254
- 7.3 The entity life cycle 257
 - Tracking changes* 259 ■ *Submitting changes* 260
 - Working with disconnected data* 263
- 7.4 Summary 266

8 *Advanced LINQ to SQL features* 267

- 8.1 Handling simultaneous changes 268
 - Pessimistic concurrency* 268 ■ *Optimistic concurrency* 269
 - Handling concurrency exceptions* 272 ■ *Resolving conflicts using transactions* 276
- 8.2 Advanced database capabilities 278
 - SQL pass-through: Returning objects from SQL queries* 278
 - Working with stored procedures* 280 ■ *User-defined functions* 290
- 8.3 Improving the business tier 294
 - Compiled queries* 294 ■ *Partial classes for custom business logic* 296 ■ *Taking advantage of partial methods* 299
 - Using object inheritance* 301
- 8.4 A brief diversion into LINQ to Entities 306
- 8.5 Summary 309

PART 4 MANIPULATING XML 311

9 *Introducing LINQ to XML* 313

- 9.1 What is an XML API? 314
- 9.2 Why do we need another XML programming API? 316
- 9.3 LINQ to XML design principles 317
 - Key concept: functional construction* 319 ■ *Key concept: context-free XML creation* 320 ■ *Key concept: simplified names* 320
- 9.4 LINQ to XML class hierarchy 323
- 9.5 Working with XML using LINQ 326
 - Loading XML* 327 ■ *Parsing XML* 329 ■ *Creating XML* 330 ■ *Creating XML with Visual Basic XML literals* 335
 - Creating XML documents* 338 ■ *Adding content to XML* 341
 - Removing content from XML* 343 ■ *Updating XML content* 344 ■ *Working with attributes* 347 ■ *Saving XML* 348
- 9.6 Summary 349

10 *Query and transform XML with LINQ to XML* 350

- 10.1 LINQ to XML axis methods 352
 - Element* 354 ■ *Attribute* 355 ■ *Elements* 356 ■ *Descendants* 357 ■ *Ancestors* 360 ■ *ElementsAfterSelf*, *NodesAfterSelf*, *ElementsBeforeSelf*, and *NodesBeforeSelf* 362 ■ *Visual Basic XML axis properties* 363
- 10.2 Standard query operators 366
 - Projecting with Select* 369 ■ *Filtering with Where* 370
 - Ordering and grouping* 372
- 10.3 Querying LINQ to XML objects with XPath 376
- 10.4 Transforming XML 378
 - LINQ to XML transformations* 378 ■ *Transforming LINQ to XML objects with XSLT* 382
- 10.5 Summary 383

11	Common LINQ to XML scenarios	385
11.1	Building objects from XML	386
	<i>Goal</i>	387 ■ <i>Implementation</i>
11.2	Creating XML from object graphs	392
	<i>Goal</i>	392 ■ <i>Implementation</i>
11.3	Creating XML with data from a database	398
	<i>Goal</i>	399 ■ <i>Implementation</i>
11.4	Filtering and mixing data from a database with XML data	406
	<i>Goal</i>	406 ■ <i>Implementation</i>
11.5	Reading XML and updating a database	411
	<i>Goal</i>	412 ■ <i>Implementation</i>
11.6	Transforming text files into XML	428
	<i>Goal</i>	428 ■ <i>Implementation</i>
11.7	Summary	432

PART 5 LINQING IT ALL TOGETHER 435

12	Extending LINQ	437
12.1	Discovering LINQ's extension mechanisms	438
	<i>How the LINQ flavors are LINQ implementations</i>	439
	<i>What can be done with custom LINQ extensions</i>	441
12.2	Creating custom query operators	442
	<i>Improving the standard query operators</i>	443 ■ <i>Utility or domain-specific query operators</i>
12.3	Custom implementations of the basic query operators	451
	<i>Refresh on the query translation mechanism</i>	452 ■ <i>Query expression pattern specification</i>
	453 ■ <i>Example 1: tracing standard query operators' execution</i>	455 ■ <i>Limitation: query expression collision</i>
	457 ■ <i>Example 2: nongeneric, domain-specific operators</i>	459 ■ <i>Example 3: non-sequence operator</i>
	461	

- 12.4 Querying a web service: LINQ to Amazon 463
 - Introducing LINQ to Amazon* 463 ■ *Requirements* 465
 - Implementation* 467
- 12.5 IQueryable and IQueryProvider: LINQ to Amazon advanced edition 474
 - The IQueryable and IQueryProvider interfaces* 474
 - Implementation* 479 ■ *What happens exactly* 480
- 12.6 Summary 481

13 *LINQ in every layer* 482

- 13.1 Overview of the LinqBooks application 483
 - Features* 483 ■ *Overview of the UI* 484 ■ *The data model* 486
 - 13.2 LINQ to SQL and the data access layer 486
 - Refresher on the traditional three-tier architecture* 487 ■ *Do we need a separate data access layer or is LINQ to SQL enough?* 488
 - Sample uses of LINQ to SQL in LinqBooks* 495
 - 13.3 Use of LINQ to XML 502
 - Importing data from Amazon* 502 ■ *Generating RSS feeds* 504
 - 13.4 Use of LINQ to DataSet 505
 - 13.5 Using LINQ to Objects 509
 - 13.6 Extensibility 509
 - Custom query operators* 509 ■ *Creating and using a custom LINQ provider* 510
 - 13.7 A look into the future 511
 - Custom LINQ flavors* 511 ■ *LINQ to XSD, the typed LINQ to XML* 513 ■ *PLINQ: LINQ meets parallel computing* 513
 - LINQ to Entities, a LINQ interface for the ADO.NET Entity Framework* 514
 - 13.8 Summary 515
- appendix: The standard query operators* 517
- resources* 523
- index* 527

bonus chapter: *Working with LINQ and DataSets*
available online only from www.manning.com/LINQinAction

14

Working with LINQ and DataSets

This chapter covers:

- LINQ to DataSet
- A refresher on DataSets
- Working with typed and untyped DataSets
- Data binding LINQ to DataSet results

In the second part of this book, we demonstrated how LINQ to Objects can be used to query in-memory objects. In this chapter, we'll cover a different scenario. We'll still query in-memory objects, but the specific objects we'll deal with here are `DataSets` and `DataTables`. We'll show you how a dedicated LINQ flavor named LINQ to DataSet elegantly solves the problem of queries over `DataSets`.

We'll start this chapter with an overview of LINQ to DataSet and a quick refresher on `DataSets`. We'll then walk you through several examples that demonstrate the kind of operations that can be performed with LINQ to DataSet.

Before being able to query `DataSets`, we need to load them with data. This is why we'll first show you how to store the results of a LINQ query in a `DataTable`. We'll then demonstrate how it's possible to query a `DataSet` without LINQ, before showing how LINQ to DataSet makes it easier to retrieve data from a `DataSet` using LINQ queries. Our last examples will show how to join tables in queries and how to deal with relationships.

We'll show you examples of these operations with untyped `DataSets` as well as typed `DataSets`. You'll notice that in comparison to untyped `DataSets`, typed `DataSets` allow us to write more reliable and type-safe LINQ queries.

The last section of this chapter will provide a summary of the custom query operators introduced by LINQ to DataSet. With this information, you'll be equipped to make the best out of LINQ to DataSet.

14.1 Overview of LINQ to DataSet

`DataSets` can be used as convenient memory-resident data stores, but until now their query features were limited. If you want to query a `DataSet` using the classical methods, you have to write expressions using a specific syntax.¹ Even if you master this syntax, the query possibilities remain far less than what LINQ offers. Also, these expressions are built using strings, which does not provide any compile-time checking for validity.

LINQ provides a unique opportunity to introduce rich query capabilities on top of the `DataSet` class and in a way that integrates with the rest of the development environment. With LINQ to DataSet, you'll write queries in your usual development language, like C# or VB.NET.

We'll show you how to write your first LINQ to DataSet code to execute in-memory queries against `DataSets` (both untyped and typed). You'll see that

¹ You'll see samples of this syntax in section 14.3.2 when we'll demonstrate how to query `DataSets` without LINQ.

LINQ to DataSet will greatly improve your experience with DataSets. You can consider that LINQ to DataSet brings to DataSets what LINQ to SQL brings to relational databases.

To give you an idea of what to expect, here is a sample LINQ to DataSet query over a typed DataSet:

```
from publisher in dataSet.Publisher
join book in dataSet.Book
  on publisher.ID equals book.Publisher
select new {
    Publisher = publisher.Name,
    Book = book.Title
};
```

As you can see, this query looks like any other LINQ query. This is what makes LINQ great: It can be used with several data sources in a consistent way. This will allow you to write LINQ queries against DataSets in no time.

In case you're not familiar with DataSets, before going further we'll provide a refresher on them.

14.2 Refresher on DataSets

This review of DataSets will allow you to discover them if you're not used to working with them. It will also allow us to introduce some vocabulary as well as the main classes we'll use through this chapter.

We'll also give you an overview of the changes Visual Studio 2008 and .NET 3.5 introduced to enable LINQ queries against DataSets.

14.2.1 DataSet use cases and features

In contrast to other typical ADO.NET components like DataReaders, DataSets are backend-agnostic and are used in a disconnected manner. The versatility of DataSets makes them useful in a range of use cases, from quick-and-dirty applications—which benefit from the simplicity of DataSets and their built-in data storage capabilities—to reporting solutions—where a discrete set of data is generated and queried in memory.

Table 14.1 lists the main features offered by DataSets and sample scenarios they enable.

Now that you have an idea why DataSets are powerful and when they're useful, let's get a better idea of what they are.

Table 14.1 DataSet's features and the scenarios they enable

DataSet's features	Enabled scenarios
A DataSet can store data in multiple tables, which can be related through foreign key relationships. A DataSet is independent of any data source.	Manipulate data from multiple sources (for example, a mixture of data from more than one database, from an XML file, and from a spreadsheet). Navigate between multiple discrete tables of results.
Constraints are enforced on data at row level. Calculated fields are supported. Multiple versions of the data are stored for each column and for each row in each table. Ongoing data edits can be accepted or rejected.	Use transactional local data store.
A DataSet can be used in disconnected mode (no permanent connection to a database is required during the object's lifetime).	Exchange data between tiers or using a web service. Perform processing on data that takes too much time to keep a database connection opened.
A DataSet can be serialized into a file or a stream, in XML or binary format.	Cache data in memory or on the disk.

14.2.2 What are DataSets?

A DataSet is an ADO.NET type that is frequently used as an in-memory data store. DataSets can be compared to memory-resident minirelational databases. A DataSet represents a complete set of data including the tables that contain the data, as well as the relationships between the tables. Inside a DataSet—much like in a relational database—there are tables, columns, relationships, constraints, views, and all the appropriate metadata that describe the DataSet's structure.

Typically a DataSet is used to represent a set of data required for a use case. Let's imagine you're working on an author's books. The data you'd deal with would likely consist of information such as the author's ID, his first name and last name, the list of books he has written, and the details about each book, such as the publisher and the book subject. Usually, this data is stored in memory using a dedicated Author business entity.

One way to represent a business entity is to use a DataSet. Figure 14.1 shows how the author and books data could be represented as a DataSet.

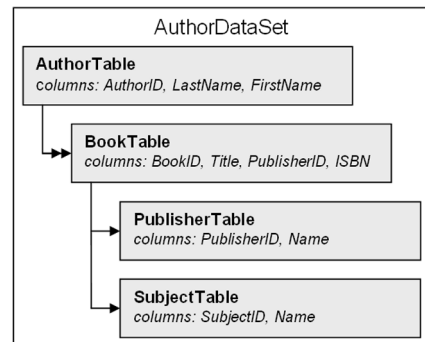


Figure 14.1 An Author entity represented as a DataSet and the four tables it contains. Later on, we'll create a simpler DataSet for our examples.

The figure depicts a DataSet that contains four tables with relationships between them. A row in AuthorTable is linked to multiple rows in BookTable. Each row in BookTable points to one row in PublisherTable and one row in SubjectTable.

This is what a typical DataSet looks like. Of course, the figure doesn't show how constraints or calculated columns can come into play. The goal here is not to give you complete coverage of DataSets but to keep the book focused on LINQ. You can find more information about DataSets in Microsoft's official documentation and in books and web sites that cover ADO.NET.

Although we can't cover everything, there are still more things we'd like you to know about DataSets that will help you to have a better understanding of the working relationship between DataSets and LINQ to DataSet. Let's dive into the DataSet class and the associated classes.

DataSet-related classes

DataSets are made available in the System.Data namespace as a set of classes. The main class is System.Data.DataSet. The class diagram in figure 14.2 shows how the different classes relate to each other.

To keep things simple, remember that a DataSet (DataSet class) is a collection of tables (DataTable class) and table relationships (DataRelation class). A table is a collection of columns (DataColumn class), rows (DataRow class), and constraints (Constraint class).

What you'll deal with in LINQ to DataSet queries are mostly DataTables and DataRows. You may also use the relationships that exist between some tables.

DataSets are usually used in two ways: as bare System.Data.DataSet objects that expose System.Data.DataTable and System.Data.DataRow objects, or as strongly typed objects that inherit from DataSet, DataTable, and DataRow. In the

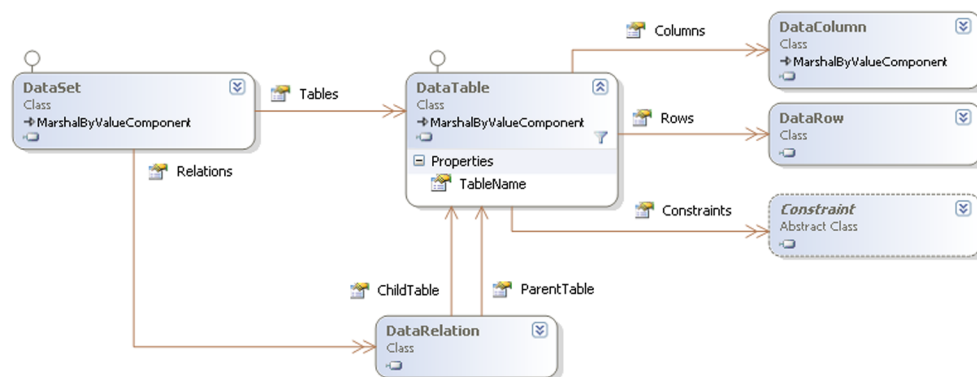


Figure 14.2 Class diagram showing the classes involved with untyped DataSets. We'll see in the next section how typed DataSets extend this model.

latter case, we'll speak of typed DataSets. Typed DataSets enable compile-time validation of column types and names, as well as IntelliSense. In general, it's better to use typed DataSets and keep untyped DataSets for simple or generic code.

This may be difficult to grasp if you aren't familiar with DataSets, but don't worry; it will make more sense when we demonstrate how to use LINQ to DataSet both with untyped and typed DataSets in sections 14.3 and 14.4. But before we can explore these two scenarios, we should tell you more about typed DataSets. This will help you to understand more precisely how our sample LINQ queries against typed DataSets work.

Typed DataSets

When you use untyped DataSets, you deal directly with a set of classes provided by the .NET Framework, namely DataSet, DataTable, and DataRow. In the case of typed DataSets, you work with a set of ad hoc classes that inherit directly from the DataSet family of classes. Each typed DataSet introduces new classes that inherit from the DataSet, DataTable, and DataRow classes.

Figure 14.3 shows the classes involved in a sample typed DataSet.

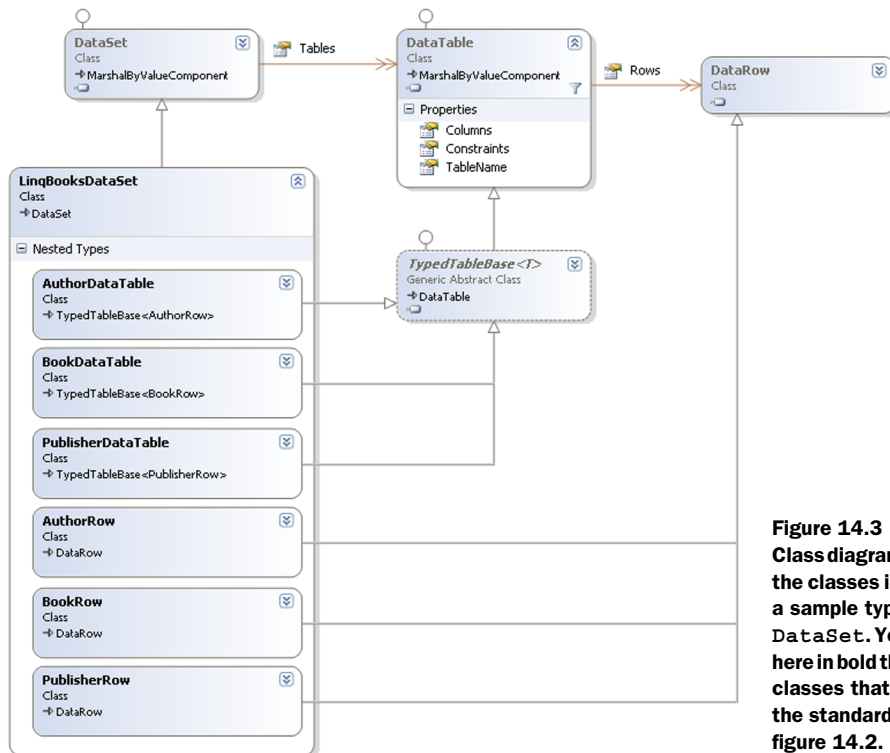


Figure 14.3
Class diagram showing the classes involved in a sample typed DataSet. You can see here in bold the custom classes that extend the standard shown in figure 14.2.

In the class diagram, you can find the main classes that were already involved in an untyped DataSet: DataSet, DataTable, and DataRow. The main class in the diagram is `LinqBooksDataSet`; this is the DataSet. Of course, this class inherits from the DataSet class. You can see that our typed DataSet contains three tables, each one represented by a specific class: `AuthorDataTable`, `BookDataTable`, `PublisherDataTable`. All these tables inherit from the `TypedTableBase<T>` class, which we'll detail in a moment. The other classes in the diagram are used to represent rows in the tables: `AuthorRow`, `BookRow`, `PublisherRow`. They all extend the `DataRow` class.

A typed DataSet derives from the DataSet class, and its nested classes derive from other standard classes. This means that you do not sacrifice any of the DataSet functionality when you use a typed DataSet instead of an untyped one. A typed DataSet remains a DataSet, but with more features.

There are several benefits when using typed DataSets, which include type checking at compile time and support from Visual Studio's IntelliSense. For example, instead of exposing the Title column of a row from a table of books, you expose the Title property of a Book object. This allows a more object-oriented approach and enables IntelliSense to display the list of a row's fields as you type. The type of each property also allows the compiler to enforce type safety.

Usually, you don't create the inherited classes by hand. A typed DataSet is typically created from an XML Schema Document (XSD). An XSD file allows you to describe the structure and constraints you want in your DataSet. C# or VB.NET code can then be generated based on the information contained in the XSD using either the XSD.exe command-line tool or Visual Studio. Both solutions generate a code file that contains the specific classes that constitute a typed DataSet. In 14.4.1, we'll demonstrate how to create a typed DataSet using Visual Studio.

Now that you have a good idea of what DataSets are, it may be useful to point out what's new in the latest versions of Visual Studio and .NET regarding DataSets. This will allow you to make the transition to Visual Studio 2008 and .NET 3.5 if you're used to working with DataSets in previous versions of the platform.

14.2.3 What's new in Visual Studio 2008 and .NET 3.5 to make LINQ queries against DataSets possible

In order to enable LINQ queries over DataSets, the code generator for typed DataSets in Visual Studio 2008 and Xsd.exe has been improved. Previous versions of Visual Studio represented a DataSet's tables as classes inherited from `System.Data.DataTable`. The tables contained in the typed DataSets created by the

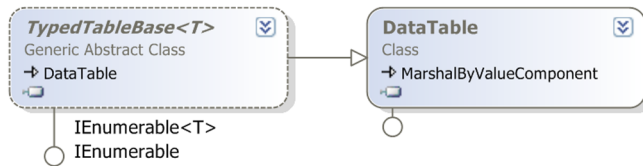


Figure 14.4 Class diagram showing how the new `System.Data.TypedTableBase<T>` class extends the `System.Data.DataTable` class by implementing `IEnumerable<T>` and `IEnumerable`.

new generator are represented using classes inherited from a new class named `System.Data.TypedTableBase<T>`.

Figure 14.4 is a class diagram that shows the `TypedTableBase<T>` class and its relation to the `DataTable` class.

The new `TypedTableBase<T>` class still inherits from `DataTable`, so it's an enhancement that doesn't break backward compatibility. `TypedTableBase<T>` adds to `DataTable` implementations of `System.Collections.Generic.IEnumerable<T>` and `System.Collections.IEnumerable`. This turns `DataTables` into queryable collections, and allows you to query the tables contained in a typed `DataSet` using LINQ.

In addition to the `TypedTableBase<T>` class and the new typed `DataSet` generator, a set of extension methods has also been added to the .NET Framework's classes. These methods facilitate the integration of `DataSets` within LINQ queries.

Now that we've shown you what `DataSets` are and when to use them, we can get back to the focus of this chapter, which is LINQ to `DataSet`. We understand that this brief introduction to `DataSets` may be difficult to follow, but as soon as we get to the code examples, we believe that the fog will lift. The `DataSet` infrastructure and the evolutions needed to support LINQ may seem complex at first. You'll see in the following sections that they're mostly transparent when writing code that combines `DataSets` and LINQ queries.

We'll now demonstrate how you can query `DataSets` with LINQ in similar ways to how you'd query other data structures. LINQ to `DataSet` includes full support for LINQ queries over both untyped and typed `DataSets`. We'll demonstrate both scenarios, starting with untyped `DataSets`.

14.3 Querying untyped DataSets

In this section, you'll see how to write your first LINQ to `DataSet` queries. Before we can query a `DataSet`, it needs to contain data. This is why we'll start by showing you

options for loading data into DataSets. We'll demonstrate how to use DataAdapters to fill a DataSet with data from a database, and then we'll demonstrate another approach that allows data from a LINQ query to be loaded into a DataSet.

Once we have some data in a DataSet, we'll show you how to write queries against it, and right before writing LINQ to DataSet code, we'll highlight the other options for querying DataSets that were available before LINQ appeared. Finally, we'll write simple LINQ to DataSet queries as well as queries that join tables.

For the moment, let's load some data into DataSets.

14.3.1 Loading data into DataSets

Several means are available to load data into a DataSet. We'll demonstrate two approaches. The first consists of providing SQL queries and using them through a DataAdapter. The second approach relies on LINQ to SQL.

Inherently, a DataSet can save and load data to and from a stream or a file. A DataSet also offers methods for serializing and deserializing data as an XML string.

Most of the time, the data you'll want to store in a DataSet comes from a relational database. But because the DataSet class was designed to be independent of any data source, it doesn't offer methods for loading data from a database and pushing changes back into the database. ADO.NET includes classes that make it easy to accomplish these tasks: DataAdapters.

Using a DataAdapter

A DataAdapter is an object that provides a bridge to retrieve and save data between a DataSet and a database. A DataAdapter connects to a database to fill a DataSet. Later on, the DataAdapter can connect back to the database to update the data there, based on operations performed in-memory on the data held in the DataSet.

Figure 14.5 shows the interactions between a DataSet, a DataAdapter, and a database.

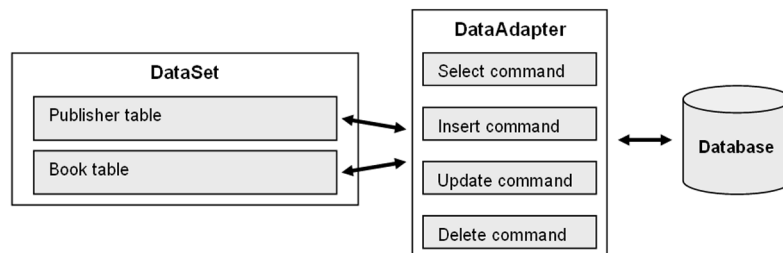


Figure 14.5 Interaction between a DataSet, a DataAdapter, and a database

A `DataAdapter` loads and persists data by means of SQL queries made against the database. In order to load data into a `DataSet`, you need to provide a `DataAdapter` with SQL queries. You'll then be able to invoke the `DataAdapter`'s `Fill` method to actually load the data.

Listing 14.1 shows a sample code snippet that demonstrates how to use a `DataAdapter`.

Listing 14.1 Loading data into a `DataSet` using a `DataAdapter`

```
void FillDataSetUsingDataAdapter(DataSet dataSet)
{
    // Create the DataAdapter
    var dataAdapter = new SqlDataAdapter(
        @"SELECT ID, Name
        FROM Publisher
        ;
        SELECT ID, Title, Subject, Publisher, Price
        FROM Book
        WHERE DATEPART(YEAR, PubDate) > 1950 ",
        Properties.Settings.Default.LinqBooksConnectionString);

    // Map the results to tables in the DataSet
    dataAdapter.TableMappings.Add("Table", "Publisher");
    dataAdapter.TableMappings.Add("Table1", "Book");

    // Execute the SQL queries and load the data into the DataSet
    dataAdapter.Fill(dataSet);
}
```

NOTE We use the `SqlDataAdapter` class because we're working with a SQL Server database. Specific `DataAdapter` classes should be used for other databases, such as `OracleDataAdapter` or `SybaseDataAdapter`. Optionally, you could use `OleDbDataAdapter`, which is not dependent on a specific DBMS.

Now that we have a method to fill a `DataSet`, we can display its content, as we do using `DataGridView` controls in figure 14.6.

Here is the code that you can use to create the `DataSet` and bind it to the `DataGridViews`:

<pre>var dataSet = new DataSet(); FillDataSetUsingDataAdapter(dataSet);</pre>	Load the DataSet
<pre>dataGridView1.DataSource = dataSet.Tables[0]; dataGridView2.DataSource = dataSet.Tables[1];</pre>	Display its content

The screenshot shows a Windows application titled "LINQ to DataSet". It has a menu bar with "Loading", "Without LINQ", "Untyped DataSet", "Typed DataSet", "CopyToDataTable", and "DataRowComparer". Below the menu is a "Test loading" button. The main area displays two DataGridViews. The first, "Publishers", has columns "ID" and "Name" and contains two rows: "4ab0856e-51f3-4..." (FunBooks) and "855cb02e-dc29-..." (Joe Publishing). The second, "Books", has columns "ID", "Title", "Subject", "Publisher", and "Price" and contains five rows of book data.

ID	Name
4ab0856e-51f3-4...	FunBooks
855cb02e-dc29-...	Joe Publishing

ID	Title	Subject	Publisher	Price
0737c167-e3d9-...	All your base are ...	c603e018-7e60-...	855cb02e-dc29-...	
b1c7670c-fdf5-4...	C# on Rails	a0e2a5d7-88c6-...	855cb02e-dc29-...	35,5000
4f3b0ac1-3746-4...	Funny Stories	a0e2a5d7-88c6-...	4ab0856e-51f3-4...	25,5500
5a361453-13ee-...	LINQ rules	a0e2a5d7-88c6-...	855cb02e-dc29-...	12,0000
09017e35-ca66-...	Bonjour mon Amour	92f10ca6-7970-4...	4ab0856e-51f3-4...	29,0000

Figure 14.6 The content of a DataSet displayed using DataGridViews

We've just used the traditional method for loading data into a DataSet. Because we now have LINQ in our toolset, it'd be a pity not to use it for this task. We'll now demonstrate how the result of a LINQ query can be stored in a DataTable.

Using LINQ to SQL to load data into DataSets

The problem with using DataAdapters is that it's a brittle approach that requires writing SQL queries. The SQL queries we write can easily become invalid, which would lead to issues at run-time. LINQ to SQL already solves these kinds of problems, so it would be excellent if we could use it to load data into DataSets.

LINQ to DataSet does not come with a built-in solution for storing the results of a LINQ query in a DataTable, but it's easy to write code that performs this task.

Let's demonstrate how listing 14.1 can be rewritten using LINQ to SQL. Listing 14.2 shows a new code snippet that uses two LINQ to SQL queries to add two DataTables to a DataSet.

Listing 14.2 Loading data into a DataSet using LINQ to SQL

```
void FillDataSetUsingLinqToSql1(DataSet dataSet)
{
    DataTable table;

    var linqBooks =
        new LinqBooks(
            Properties.Settings.Default.LinqBooksConnectionString);
```

**Prepare the
LINQ to SQL
DataContext**

```

var publisherQuery =
    from publisher in linqBooks.Publisher
    select new { publisher.ID, publisher.Name };

var bookQuery =
    from book in linqBooks.Book
    where book.PubDate.Value.Year > 1950
    select new {
        book.ID, book.Title, book.Subject, book.Publisher,
        Price = book.Price.HasValue ? book.Price.Value : 0
    };

table = new DataTable();
table.Columns.Add("ID", typeof(Guid));
table.Columns.Add("Name", typeof(String));

foreach (var publisher in publisherQuery)
    table.LoadDataRow(
        new Object[] {publisher.ID, publisher.Name}, true);

dataSet.Tables.Add(table);

table = new DataTable();
table.Columns.Add("ID", typeof(Guid));
table.Columns.Add("Title", typeof(String));
table.Columns.Add("Subject", typeof(Guid));
table.Columns.Add("Publisher", typeof(Guid));
table.Columns.Add("Price", typeof(Decimal));

foreach (var book in bookQuery)
    table.LoadDataRow(new Object[] {book.ID, book.Title,
        book.Subject, book.Publisher, book.Price}, true);

dataSet.Tables.Add(table);
}

```

**Query the
Publisher and
Book tables**

1 Nullable
types aren't
supported

**Prepare the
Publisher
DataTable,
load data into
it, and add it**

**Prepare the Book
DataTable, load
data into it, and
add it**

NOTES Here we use LINQ to SQL queries as the data sources. Any LINQ query can be used to achieve the same result. The data can be stored in XML documents and queried with LINQ to XML for example.

DataSets do not support nullable types. This is why we must test whether the Price property of each book we retrieve from the database has a value and we use 0 if it doesn't ❶.

Early releases of LINQ provided two query operators to perform the same kind of operation—LoadSequence and ToDataTable—but they've been removed in later

releases of LINQ, and aren't included in .NET 3.5. But they have been resurrected by Andrew Conrad from Microsoft.

NOTE The source code for the `ToDataTable` and `LoadSequence` query operators is available at <http://blogs.msdn.com/aconrad/archive/2007/09/07/science-project.aspx>.

Note that they've been renamed `CopyToDataTable` instead of `ToDataTable` and `LoadSequence`, but we've decided to keep the original names to avoid confusion with the existing `CopyToDataTable` methods from `System.Data.DataTableExtensions`.

Listing 14.3 shows how listing 14.2 can be simplified thanks to `ToDataTable`.

Listing 14.3 Loading data into a DataSet using the `ToDataTable` query operator

```
void FillDataSetUsingLinqToSql2(DataSet dataSet)
{
    DataTable table;

    // Prepare the LINQ to SQL DataContext
    var linqBooks =
        new LinqBooks(
            Properties.Settings.Default.LinqBooksConnectionString);

    // Query the Publisher table
    var publisherQuery =
        from publisher in linqBooks.Publisher
        select new { publisher.ID, publisher.Name };
    // Query the Book table
    var bookQuery =
        from book in linqBooks.Book
        where book.PubDate.Value.Year > 1950
        select new {
            book.ID, book.Title, book.Subject, book.Publisher,
            book.PageCount,
            Price = book.Price.HasValue ? book.Price.Value : 0
        };

    dataSet.Tables.Add(publisherQuery.ToDataTable());
    dataSet.Tables.Add(bookQuery.ToDataTable());
}
```

Execute the queries
and load the data
into the DataSet

Now that we have some data in our `DataSet`, we can query it. We'll first look at how we can query `DataSets` without LINQ, and then with LINQ to `DataSet` to show why LINQ is the best option.

14.3.2 Querying DataSets without LINQ

Even without resorting to LINQ, DataSets offer some query features. A first option you can use to query data within a DataSet is use the `DataTable.Select` method. This method can be used to retrieve an array of all the `DataRow` objects that match filter criteria, optionally in a specified sort order.

Here are some examples:

```
DataRow[] publishers = dataSet.Tables[0].Select("LEN(Name) > 5");
DataRow[] books =
    dataSet.Tables[1].Select(
        "(Price > 15) AND (Title LIKE '*i*)'", "Title DESC");
```

Another option for querying DataSets is made available through the `System.Data.DataView` class. A `DataView` can be used to sort and filter a `DataTable`:

```
dataGridView1.DataSource = new DataView(dataSet.Tables[0],
    "LEN(Name) > 5", String.Empty, DataViewRowState.Unchanged);
dataGridView2.DataSource = new DataView(dataSet.Tables[1],
    "(Price > 15) AND (Title LIKE '*i*)'",
    "Title DESC",
    DataViewRowState.Unchanged);
```

One advantage of `DataViews` compared to the `DataTable.Select` method is that they retain the metadata from the `DataTables`. This allows a direct binding to a `DataGridView`, as in the previous code snippet. This results in the display shown in figure 14.7.

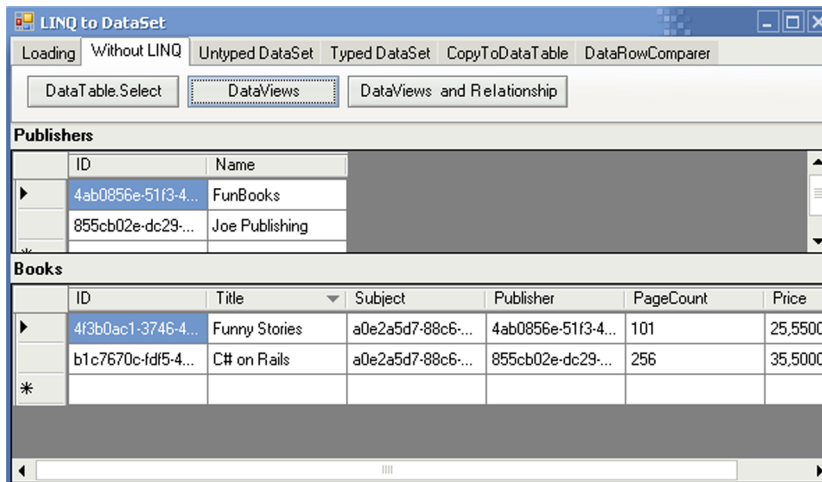


Figure 14.7 The content of `DataTables` filtered using `DataViews`. Only the books matching our criteria are displayed this time.

The techniques offered by `DataTable.Select` and `DataView` can be used with relationships. Let's assume there is a relationship between the two tables in our `DataSet`.

```
dataSet.Relations.Add("PublisherBooks",
    dataSet.Tables[0].Columns["ID"],
    dataSet.Tables[1].Columns["Publisher"]);
```

**Create a relationship
between a publisher
and its books**

```
dataGridView1.DataSource = new DataView(dataSet.Tables[0],
    "COUNT(CHILD(PublisherBooks).Title) > 0",
    String.Empty,
    DataRowState.Unchanged);
```

**Display only
publishers that
have a book in
the table**

Although `DataTables` and `DataViews` offer useful and flexible ways to filter and sort data, these features aren't as powerful as LINQ queries. We'll now start working with LINQ to `DataSet` and write LINQ queries over `DataSets` and `DataTables`.

14.3.3 Querying untyped DataSets using LINQ to DataSet

In the previous section, we showed that `DataSets` can be queried without resorting to LINQ. This works to a degree. Before LINQ to `DataSet`, queries over `DataSets` were restricted to a limited set of operators and provided no compile-time checking whatsoever. LINQ to `DataSet` enables a general mechanism for rich querying over `DataSet` objects, which allows the full power of LINQ and the .NET Framework to be utilized when writing queries.

We'll now demonstrate how LINQ to `DataSet` can be used to query `DataSets` using the same features and syntax LINQ offers for other data stores. This section focuses on untyped `DataSets`. We'll start with a simple query and continue with queries joining tables and using relationships. In section 14.4.3, we'll do the same operations but with typed `DataSets`.

Writing a simple query

In order to get started with LINQ to `DataSet` queries, let's focus on a simple query.

The code in listing 14.4 shows how to query a `DataTable` containing books.

C#

Listing 14.4 Querying an untyped DataSet with LINQ to DataSet

```
var dataSet = new DataSet();
FillDataSetUsingLinqToSql2(dataSet);

DataTable bookTable = dataSet.Tables[1];
```

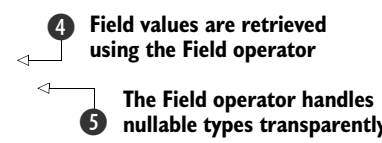
**1 Fill a DataSet
using LINQ to SQL**

**2 Retrieve one
DataTable
contained in
the DataSet**

```
var filteredBooks =
    from book in bookTable.AsEnumerable()
    where book.Field<String>("Title").StartsWith("L")
```

**3 Convert the DataTable into
an IEnumerable<DataRow>**


```
select new {  
    Title = book.Field<String>("Title"),  
    Price = book.Field<Decimal?>("Price")  
};  
  
dataGridView1.DataSource = filteredBooks.ToList();
```



4 Field values are retrieved using the Field operator

5 The Field operator handles nullable types transparently

Several things are at play in the listing. Let's describe them one by one:

- Steps ❶ and ❷ are used to prepare a `DataTable` and load it using LINQ to SQL with data coming from a database. This uses a technique we introduced in section 14.3.1.
- LINQ works on sources that are `IEnumerable<T>` or `IQueryable<T>`. As the `DataTable` class doesn't implement either interface, you must call the `AsEnumerable` operator ❸ and use the returned `IEnumerable<DataRow>` object as the source in LINQ queries.
- The elements we iterate over are `DataRow` objects. In order to retrieve values, we use the `Field` query operator ❹. The `Field` operator requires the type of the column to be specified as a type argument. The parameter it takes is the name of the column. More information about the `Field` operator is available in section 14.6.1.
- Note how nullable types are handled automatically by the `Field` operator ❺. We don't need to add special processing for the `Price` field to test for `DBNull` as is usually the case for nullable fields. More on this in section 14.6.1 when we cover the `Field` operator.

NOTE No additional namespaces need to be imported to use the `Field` and `AsEnumerable` query operators or other LINQ to DataSet features. Everything is available in the `System.Data` namespace. However, in order to use LINQ to DataSet, you need to reference the `System.Data.DataSetExtensions.dll` assembly, where its query operators and classes are defined.

`Field` is an extension method provided by the `System.Data.DataRowExtensions` class and `AsEnumerable` is an extension method provided by the `System.Data.DataTableExtensions` class. You don't need to use these classes directly. Importing the `System.Data` namespace is enough to get access to the `Field` and `AsEnumerable` query operators.

LINQ to DataSet code in VB.NET is similar to code in C#. Here is how the LINQ to DataSet query from listing 14.4 can be written using VB.NET:

VB.NET

```
From book In bookTable.AsEnumerable() _
Where book.Field(Of String)("Title").StartsWith("L") _
Select _
    Title = book.Field(Of String)("Title"), _
    Price = book.Field(Of Decimal)("Price") _
```

In comparison to C#, VB.NET offers a specific syntax that can make queries shorter. In VB.NET, an exclamation mark (also known as *bang*, *pling*, or the *dictionary lookup operator* in VB) can be used in lieu of the `Field` operator. Here is the same query as we just saw, but formulated with the simplified syntax:

VB.NET

```
From book In bookTable.AsEnumerable() _
Where book!Title.StartsWith("L") _
Select Title = book!Title, Price = book!Price
```

This syntactic sugar shortens the code and removes the need for specifying the data types of the fields. The latter query can even be simplified because we keep the name of the fields as the names of the properties in the select clause:

VB.NET

```
From book In bookTable.AsEnumerable() _
Where book!Title.StartsWith("L") _
Select book!Title, book!Price
```

NOTE An interesting observation is that Visual Basic's support for late binding allows queries against untyped DataSets to be easier to read than their C# counterparts. In section 14.4, we'll discuss how typed DataSets address this issue making readability significantly better for both languages.

Now that you've written your first LINQ to DataSet queries, you should start to see how this technology allows you to query DataSets using the LINQ syntax and power you're now becoming familiar with.

Let's move on to richer queries. In the next sections we'll demonstrate how to write queries that join tables, with or without a predefined relationship.

Joining DataTables

We've just seen a simple example, but LINQ to DataSet supports all the LINQ operators that make sense over a DataTable or a sequence of DataRow objects. One common request for DataSets is support for joins across DataTables. This is now possible with LINQ thanks to the join operators.

Just like you'd do with LINQ to Objects, you can use the `join` clause in a LINQ to DataSet query expression to join tables. Figure 14.8 shows an example of the result we want to achieve.

In the figure, you can see that data from both the Publisher and Book tables have been projected into one collection.

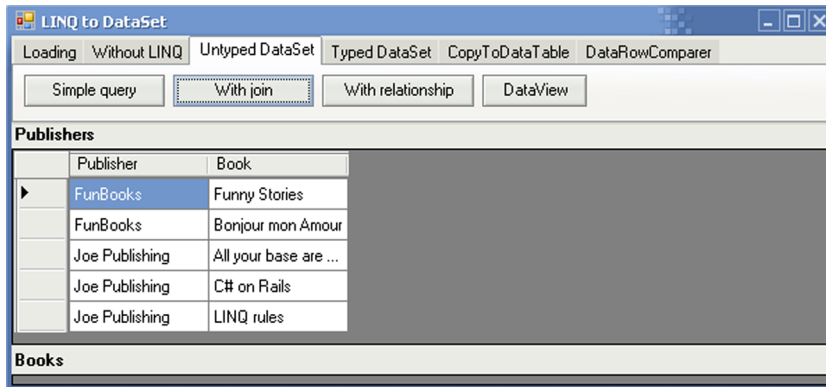


Figure 14.8 Book and publisher tables projected into one table through a LINQ to DataSet join

Listing 14.5 is a sample code snippet that shows how to join the Book and Publisher tables.

Listing 14.5 Joining untyped tables in a LINQ to DataSet query

```
var dataSet = new DataSet();
FillDataSetUsingLinqToSql2(dataSet);

DataTable publisherTable = dataSet.Tables[0];
DataTable bookTable = dataSet.Tables[1];

var publisherBooks =
    from publisher in publisherTable.AsEnumerable()
    join book in bookTable.AsEnumerable()
      on publisher.Field<Guid>("ID")
      equals book.Field<Guid>("Publisher")
    select new {
        Publisher = publisher.Field<String>("Name"),
        Book = book.Field<String>("Title")
    };

dataGridView1.DataSource = publisherBooks.ToList();
```

❶ Join the Publisher and Book tables using the publisher ID for the join criterion

❷ Retrieve information from both tables

In the listing, we use a join clause ❶ to establish a master-details relationship between a publisher and its books. We then project publisher names and book titles into an anonymous type ❷. The result is a collection of all the book titles with the associated publisher's name, as seen in figure 14.8.

Instead of re-creating the relationship in every query, we could use a feature provided by DataSets. A DataSet can contain information about the relationships between its tables. We'll use this the feature in the following section.

Working with table relationships

In listing 14.5, we joined the Publisher and Book tables. In situations like this, it's possible to let the DataSet know that there is a relationship between two tables. This way, instead of using a join clause in your query, you can use the DataRow.GetChildRows method.

Listing 14.6 is a code snippet equivalent to listing 14.5 that uses this technique.

Listing 14.6 LINQ to DataSet query on an untyped DataSet using relationships

```
dataSet.Relations.Add("PublisherBooks",  
    publisherTable.Columns["ID"], bookTable.Columns["Publisher"]);  
  
var publisherBooks =  
    from publisher in publisherTable.AsEnumerable()  
    from book in publisher.GetChildRows("PublisherBooks")  
    select new {  
        Publisher = publisher.Field<String>("Name"),  
        Book = book.Field<String>("Title")  
    };  
};
```

Query the
DataTables

Create a
relationship
between a
publisher
and its
books

With DataSet relationships, the code is simplified and easier to read. In the version of the query that uses DataRow.GetChildRows, the fields forming the relationships aren't specified. This information is part of the relationship present in the DataSet. Also, the relationship is named, which makes it easier to understand how the tables are joined. Note that it's also possible to navigate the tables in the opposite direction using DataRow.GetParentRow or DataRow.GetParentRows.

This is all we'll demonstrate for untyped DataSets because you already know the rest. The queries you can write using LINQ to DataSet follow the same syntax and rules as with the other flavors of LINQ. You can write query expressions and use the standard query operators with DataSets as you would with other in-memory objects.

We'll now focus on typed DataSets. We'll show you how to write the same kind of queries you've just seen with untyped DataSets, but this time with typed DataSets. This will allow you to see how it's easier and safer to write LINQ queries against typed DataSets than against untyped DataSets.

14.4 Querying typed DataSets

Since typed DataSets are still DataSets, the LINQ to DataSet queries you've seen in the previous section about untyped DataSets also apply to typed DataSets. Also, the capability of typed DataSets to expose table fields as strongly typed properties enables us to write improved and simplified queries.

Let's take a second look at the query we wrote for untyped DataSets as part of listing 14.4.

```
from book in bookTable.AsEnumerable()  
where book.Field<String>("Title").StartsWith("L")  
select new {  
    Title = book.Field<String>("Title"),  
    Price = book.Field<Decimal>("Price")  
};
```

While LINQ to DataSet is a powerful tool when used as shown, the code suffers from a number of limitations:

- Field access is untyped by default (accessing a DataRow's field returns an untyped object). Use of the Field operator is required to handle type casting and null values.
- Use of the Field operator clutters the code with types and column names in quotes.
- Column access is done in a late-bound way, which prevents the compiler from checking column names at compile-time. Column names are evaluated at run-time only, which can result in run-time errors.

NOTE While VB.NET offers a simplified syntax (see section 5.3.3), it still suffers from the lack of compile-time checking due to the use of late binding.

If the schema of the DataSet is known at design-time then typed DataSets provide a much better experience when using LINQ. Rows in typed DataSets have typed members for each column, which makes access much easier. Additionally, typed DataSets have properties for easy access to the various tables they contain.

In this section, after we show you how to create a typed DataSet, we'll compare the queries we wrote for untyped DataSets with equivalent queries for typed DataSets. This comparison should make it obvious that LINQ to DataSet and typed DataSets work well together to enable strongly typed queries over an in-memory relational data store.

14.4.1 *Generating a typed DataSet*

Before being able to query a typed DataSet, you need to generate one. Let's review the steps required to create a typed DataSet based on the data model of our running LinqBooks example.

The first step consists in adding a new DataSet to a Visual Studio project. This can be achieved by clicking the Project menu and selecting Add New Item.... The Add New Item dialog box appears, as shown in figure 14.9.

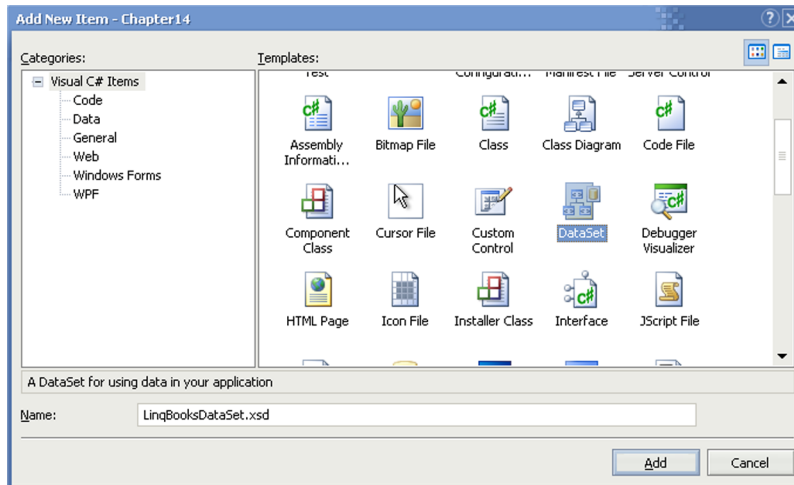


Figure 14.9 Dialog for adding a new DataSet to a project

Select DataSet and name the item LinqBooksDataSet.xsd.

The DataSet Designer is then available to visually create our typed DataSet, as shown in figure 14.10.

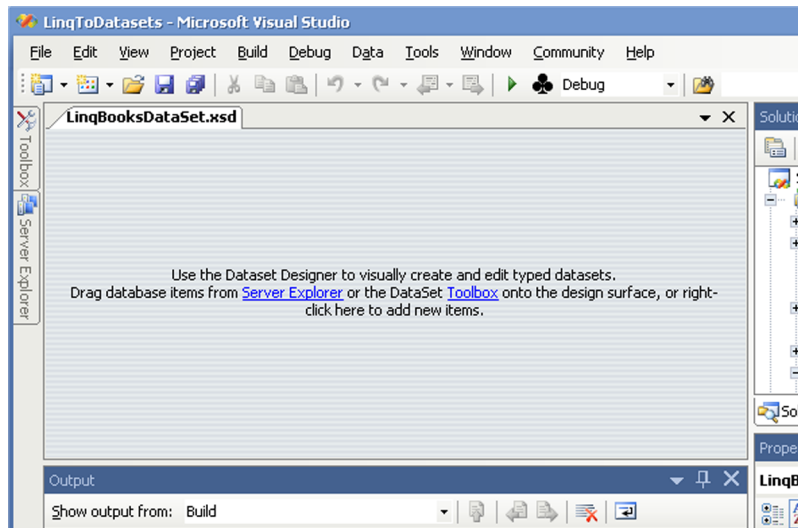


Figure 14.10 Editing a new typed DataSet using the DataSet Designer after adding it to a project

We'll select some tables from our database and add them to the DataSet. Open the Server Explorer and add a connection to the LinqBooks database if you don't have one yet.

Figure 14.11 shows the Server Explorer with a connection to the LinqBooks database.

Drag and drop the Book and Publisher tables onto the design surface, as in figure 14.12.

Note that the relationships are created automatically between the tables in the DataSet based on the information contained in the database.

When you save the new DataSet, Visual Studio automatically creates a C# or VB.NET code file. You can find it below the LinqBooksDataSet.xsd file in the Solution Explorer. It's named LinqBooksDataSet.Designer.cs or LinqBooksDataSet.Designer.vb.

The code generated for our DataSet looks like listing 14.7.

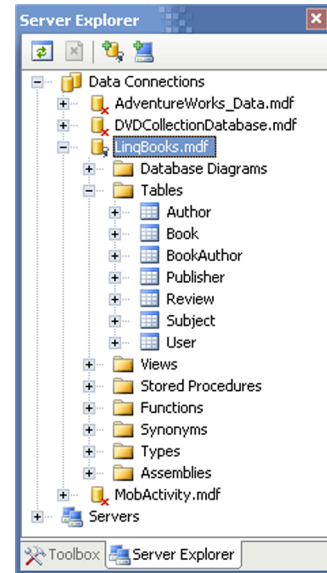


Figure 4.11 Visual Studio's Server Explorer showing the LinqBooks database and its tables

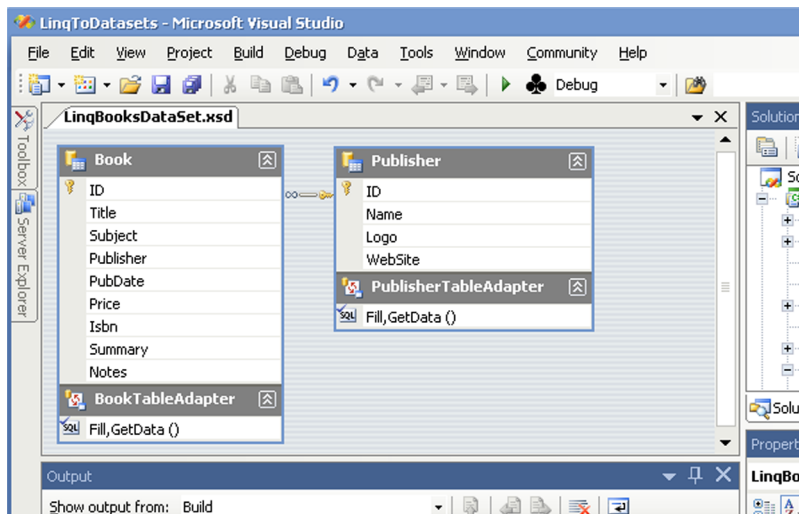


Figure 14.12 The Book and Publisher tables designed using the DataSet Designer

Listing 14.7 Sample code generated for the typed DataSet we created and designed (LinqBooksDataSet.Designer.cs)

```
//-----
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.1433
//
//     Changes to this file may cause incorrect behavior and will
//     be lost if the code is regenerated.
// </auto-generated>
//-----

#pragma warning disable 1591

namespace LinqInAction.Chapter14 {
    /// <summary>
    /// Represents a strongly typed in-memory cache of data.
    /// </summary>
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "System.Data.Design.TypedDataSetGenerator", "2.0.0.0")]
    [global::System.Serializable()]
    [global::System.ComponentModel.DesignerCategoryAttribute(
        "code")]
    [global::System.ComponentModel.ToolboxItem(true)]
    [global::System.Xml.Serialization.XmlSchemaProviderAttribute(
        "GetTypedDataSetSchema")]
    [global::System.Xml.Serialization.XmlRootAttribute(
        "LinqBooksDataSet")]
    [global::System.ComponentModel.Design.HelpKeywordAttribute(
        "vs.data.DataSet")]
    public partial class LinqBooksDataSet :
        global::System.Data.DataSet
    {
        private BookDataTable tableBook;
        private PublisherDataTable tablePublisher;
        private global::System.Data.DataRelation
            relationFK_Book_Publisher;
        private global::System.Data.SchemaSerializationMode
            _schemaSerializationMode =
                global::System.Data.SchemaSerializationMode.IncludeSchema;
        ...
    }
}
```

This code file contains the classes for our typed DataSet, such as LinqBooksDataSet, BookDataTable, PublisherDataTable, BookRow, and PublisherRow.

Our typed DataSet is now complete. Before being able to query it using LINQ to DataSet, we need to load data into it.

14.4.2 Loading data into typed DataSets

As with untyped DataSets, we have several options for loading data into typed DataSets. We'll demonstrate two options: using TableAdapters and using LINQ to SQL.

Using TableAdapters

When we designed our typed DataSet, in addition to tables, Visual Studio created a TableAdapter for each table. These TableAdapters are similar to the DataAdapters we used in section 14.3.1. The difference is that the generated TableAdapters are strongly typed.

Here is how to use the TableAdapters to load data in our typed DataSet:

```
LinqBooksDataSet dataSet = new LinqBooksDataSet();
new LinqBooksDataSetTableAdapters.PublisherTableAdapter()
    .Fill(dataSet.Publisher);
new LinqBooksDataSetTableAdapters.BookTableAdapter()
    .Fill(dataSet.Book);
```

The generated TableAdapters already contain the SQL queries required to retrieve the data from the database.

If you prefer to stick to LINQ, you'll be happy to know that it's another option you can use, as you'll see next.

Using LINQ to SQL to load data into typed DataSets

In section 14.3.1, we used LINQ to SQL to load data into a DataTable. In the case of typed DataSets, DataTables already exist in the DataSets. For example, our LinqBooksDataSet class has Publisher and Book properties that point to the DataTables available in our typed DataSet. This means that the code for loading data from a LINQ query into a typed DataSet is simpler than the code for an untyped DataSet.

Listing 14.8 shows a sample method that loads data into our typed LinqBooksDataSet.

Listing 14.8 Loading data into a typed DataSet using LINQ to SQL

```
void FillDataSetUsingLinqToSql1(LinqBooksDataSet dataSet)
{
    var linqBooks =
        new LinqBooks(

        Properties.Settings.Default.LinqBooksConnectionString);

    var publisherQuery =
        from publisher in linqBooks.Publisher
        select new { publisher.ID, publisher.Name };
}
```

Prepare the
LINQ to SQL
DataContext


Query the
tables

```

var bookQuery =
    from book in linqBooks.Book
    where book.PubDate.Value.Year > 1950
    select new {
        book.ID, book.Title, book.Subject, book.Publisher,
        Price = book.Price.HasValue ? book.Price.Value : 0
    };

foreach (var publisher in publisherQuery)
{
    dataSet.Publisher.AddPublisherRow(
        publisher.ID, publisher.Name, null, null);
}
foreach (var book in bookQuery)
{
    dataSet.Book.AddBookRow(book.ID, book.Title, book.Subject,
        dataSet.Publisher.FindByID(book.Publisher),
        DateTime.MinValue, book.Price, 0, null, null, null);
}
}

```


Query the tables

Execute the queries and load the data into the DataSet

NOTE Like the code for untyped DataSets, this code can be used to load data coming from any LINQ query, not just from a LINQ to SQL query.

Just as we proposed a second way to load data into an untyped DataSet in section 14.3.1, let's rewrite listing 14.8 using the LoadSequence query operator.² The rewritten code is shown in listing 14.9.

Listing 14.9 Loading data into a typed DataSet using LINQ to SQL and the LoadSequence operator

```

void FillDataSetUsingLinqToSql1(LinqBooksDataSet dataSet)
{
    var linqBooks =
        new LinqBooks(
            Properties.Settings.Default.LinqBooksConnectionString);

    var publisherQuery =
        from publisher in linqBooks.Publisher
        select new { publisher.ID, publisher.Name };
    var bookQuery =
        from book in linqBooks.Book
        where book.PubDate.Value.Year > 1950
        select new {
            book.ID, book.Title, book.Subject, book.Publisher,

```

² See <http://blogs.msdn.com/aconrad/archive/2007/09/07/science-project.aspx>

```

        book.PageCount,
        Price = book.Price.HasValue ? book.Price.Value : 0
    };

    publisherQuery.LoadSequence(dataSet.Publisher, null);
    bookQuery.LoadSequence(dataSet.Book, null);
}

```

Equipped with these techniques for loading data into a typed DataSet, you can now move on to actually querying typed DataSets. In section 14.3.2, we showed you how to query untyped DataSets without using LINQ. The same technique can be used with typed DataSets in the same way, so we won't rehash it here. Instead, we'll focus on querying typed DataSets with LINQ to DataSet.

14.4.3 Querying typed DataSets using LINQ to DataSet

Querying a typed DataSet using LINQ to DataSet is not fundamentally different than querying an untyped DataSet. Here we'll focus on the differences between queries on untyped and typed DataSets.

We'll cover the same operations as in section 14.3.3, starting with a simple query and then demonstrating how to join tables with and without relationships.

Simple query

Let's compare the simple query we wrote in section 14.3.3 with an equivalent one against our typed DataSet.

Here is our earlier query, using an untyped DataSet:

```

from book in dataSet.Tables[0].AsEnumerable()
where book.Field<String>("Title").StartsWith("L")
select new {
    Title = book.Field<String>("Title"),
    Price = book.Field<Decimal>("Price")
};

```

And here is the same query using a typed DataSet:

```

from book in dataSet.Book
where book.Title.StartsWith("L")
select new { book.Title, book.Price };

```

As you can see, the second query is significantly simpler. In addition to the benefits that we presented in section 14.3, typed DataSets also make LINQ queries much easier to formulate and read in comparison to queries against untyped DataSets.

Table 14.2 is a summary of the key differences that a LINQ query against a typed DataSet presents compared to a LINQ query on an untyped DataSet.

Table 14.2 Benefits of typed DataSets compared to untyped DataSets

Differences	Benefits
Access to the tables contained in the DataSet is possible through members exposed by the DataSet.	Compile-time check on the table name IntelliSense
No need to use AsEnumerable.	Shorter and simpler code
No need to use the Field operator or to specify the types of fields.	Compile-time check on the table name IntelliSense Shorter and simpler code

The same benefits apply for all LINQ to DataSet queries over typed DataSets. This is also the case when joining tables, as you'll see next.

Joining tables

In order to demonstrate how to join tables in a query over a typed DataSet, listing 14.10 shows some code equivalent to that in listing 14.5.

Listing 14.10 Joining untyped tables in a LINQ to DataSet query

```

LinqBooksDataSet dataSet = new LinqBooksDataSet();
FillDataSetUsingLinqToSql2(dataSet);

var query =
    from publisher in dataSet.Publisher
    join book in dataSet.Book
      on publisher.ID equals book.Publisher
    select new {
        Publisher = publisher.Name,
        Book = book.Title
    };

dataGridView1.DataSource = query.ToList();

```

Load a DataSet

Query the DataTables

Display the results

You can see that the code is simpler with typed DataSets, compared to that required with untyped DataSets.

Let's do the same comparison with the query that uses relationships.

Working with relationships

In our example with untyped DataSets, we created a relationship between the Publisher and Book tables. Remember that the relationship was automatically defined when we designed the typed DataSet in section 14.4.1. The code generator also created methods for each relationship defined in the typed DataSet: The

`PublisherRow.GetBookRows` method is one of them. This method can be used to navigate from a publisher to its books.

Again, let's compare the code we wrote for untyped `DataSets` in listing 14.6 to the code we can write for typed `DataSets`. Here is the code with an untyped `DataSet`:

```
from publisher in publisherTable.AsEnumerable()
from book in publisher.GetChildRows("PublisherBooks")
select new {
    Publisher = publisher.Field<String>("Name"),
    Book = book.Field<String>("Title")
};
```

Here is the same code, but with a typed `DataSet`:

```
from publisher in dataSet.Publisher
from book in publisher.GetBookRows()
select new {
    Publisher = publisher.Name,
    Book = book.Title
};
```

Table 14.3 sums up the benefits of a LINQ query over a typed `DataSet` using relationships in comparison to the equivalent query on an untyped `DataSet`.

Table 14.3 Benefits of using typed `DataSets` instead of untyped `DataSets` in a LINQ query

Differences	Benefits
Typed <code>DataRows</code> expose a method for each relationship to provide access to related rows.	Compile-time check on the table name IntelliSense
Relationships are predefined in the <code>DataSet</code> and don't have to be redefined using code	Simpler code

Thanks to this section and the previous one, you should now be able to start writing your own LINQ to `DataSet` queries. In addition, the comparison between the queries written for untyped `DataSets` and the queries written for typed `DataSets` should help you to choose between both kinds of `DataSets`.

So far, you've seen how great LINQ is for querying various sorts of data sources. However, querying is just one of the operations commonly encountered when dealing with data. Another common operation is *data binding*. We'll now see how LINQ to `DataSet` supports data binding with `CopyToDataTable` and `AsDataView`.

14.5 Binding LINQ to `DataSet` query results to UI controls

At this stage, you may have noticed that LINQ to `DataSet` suffers from one major weak point: GUI binding. Using LINQ to `DataSet` the way we did in the previous

sections comes with a disadvantage: LINQ to DataSet queries allow one-way projections only. No round-trips are possible with the data source. This means that the source DataSet can't be updated directly when the data is edited in DataGridViews or other controls. It also means that updates performed in the source DataSet aren't reflected in the query results and the controls.

Solutions to these problems come in the form of two query operators—CopyToDataTable and AsDataView—provided by LINQ to DataSet. The first allows us to copy the results of a LINQ to DataSet query into a DataTable. The second can be used for two-way data binding of LINQ to DataSet query results. Let's first introduce CopyToDataTable.

14.5.1 Using CopyToDataTable to move LINQ to DataSet results into a DataTable

CopyToDataTable is a query operator that can be used to convert the results of LINQ to DataSet queries into a DataTable. The advantage of doing this is that the new DataTable can be bound to graphical controls and edited. This allows the results of your LINQ to DataSet queries to be updated and used wherever a DataTable is required, which in itself makes for an interesting feature.

The most frequent use case for CopyToDataTable is when results need to be merged with the source DataTable. This can be used to propagate the updates to a database with a DataAdapter, for instance. In this scenario, the results of the LINQ to DataSet query are not linked to the source DataSet, but they can still be round-tripped to a database.

Here is the typical use of the CopyToDataTable operator:

- 1 Data is loaded from a database into a DataSet.
- 2 The DataSet is queried using LINQ to DataSet, which allows all the powerful LINQ operations such as joins, grouping, and sorting.
- 3 The results of the LINQ query are stored in a DataTable thanks to CopyToDataTable.
- 4 The content of the DataTable is edited.
- 5 The DataTable content is merged back into the source DataTable.
- 6 The data is used to update the database using a DataAdapter object.

The first step can be achieved using the techniques we demonstrated in sections 14.3.1 and 14.4.2.

Here is an example of how to achieve the second and third steps:

```
var books =  
    from book in dataSet.Book  
    where book.Title.Contains("a")  
    orderby book.Title  
    select book;  
  
dataGridView2.DataSource = books.CopyToDataTable();
```

Now that the results of the query are bound to a `DataGridView`, they can be edited. This is the fourth step of our scenario. Once this is done, the updated data can be merged back into the original `DataSet`, as described in the fifth step:

```
DataTable dataTable = (DataTable)dataGridView2.DataSource;  
dataSet.Book.Merge(dataTable);
```

Finally, you can update the database using the `Update` method of a `DataAdapter` or simply deal with the changes in any other way. In the following code snippet, we look at the changes that have been performed:

```
DataTable changesTable = dataSet.Book.GetChanges();  
if (changesTable == null || changesTable.Rows.Count < 1)  
{  
    MessageBox.Show("No changes");  
    dataGridView1.DataSource = null;  
}  
else  
{  
    var changes =  
        from change in changesTable.AsEnumerable()  
        select new {  
            State = change.RowState,  
            OriginalTitle = change.Field<String>(  
                "Title", DataRowVersion.Original),  
            NewTitle = change.RowState != DataRowState.Deleted ?  
                change.Field<String>("Title", DataRowVersion.Current) :  
                String.Empty  
        };  
  
    dataGridView1.DataSource = changes.ToList();  
}
```

When this is executed, a display like the one in figure 14.13 can result if you delete one book and update another.

Now that you've seen how to use `CopyToDataTable` and why it's useful, let's focus on how it works.

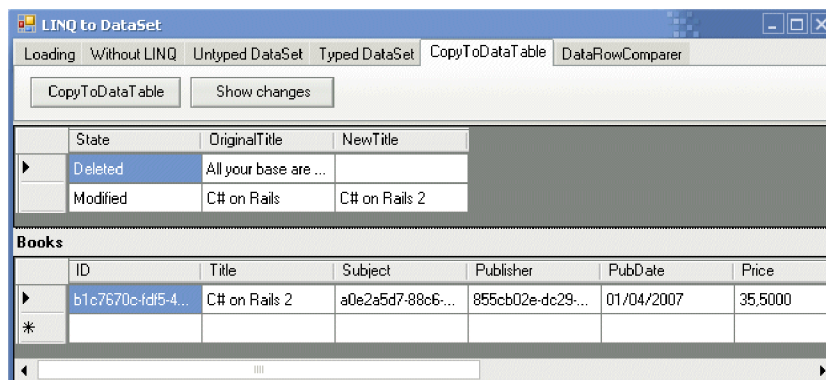


Figure 14.13 Sample display obtained with CopyToDataTable used to detect changes in a DataSet

LINQ to DataSet queries return a sequence of DataRow objects. CopyToDataTable moves these into a DataTable. Two overloads of CopyToDataTable are available:

```
DataTable CopyToDataTable<T>(this IEnumerable<T> source)
    where T: DataRow;
void CopyToDataTable<T>(this IEnumerable<T> source,
    DataTable table, LoadOption options)
    where T: DataRow;
```

The first version of CopyToDataTable returns a new DataTable containing copies of the DataRow objects contained in the source sequence. The second version of CopyToDataTable loads the DataRow objects from the source sequence into an existing DataTable. The parameter of type LoadOption can be used to control how the values from the data source will be applied to existing rows in the DataTable. This parameter specifies how changes are registered and what happens to row versions.

The schema of the destination table is based on the schema of the first DataRow in the source sequence. For a typed DataTable, types are not preserved. The data and schema are transferred, but the resulting rows of the output table will not be of the typed DataRow's type.

NOTE The CopyToDataTable query operator is an extension method provided by the System.Data.DataTableExtensions class. You don't need to use this class directly. Referencing System.Data.DataSetExtensions.dll and importing the System.Data namespace is all you need to get access to CopyToDataTable.

We've introduced the `CopyToDataTable` query operator and shown how you can bind LINQ to DataSet query results to UI components to reflect updates. We'd like to present another query operator named `AsDataView`. It enables richer data binding scenarios, namely *two-way data binding*.

14.5.2 Two-way data binding with `AsDataView`

Data binding is the process that establishes a connection between application UI and data. Basic data binding is used to easily display data in graphical components, like the Windows Forms `DataGridView` control. If the binding has the correct settings and the data provides the proper notifications then when the data changes its value, the elements that are bound to the data reflect changes automatically. If the graphical components also allow you to edit the data and your changes are reflected in the data source, this becomes two-way data binding.

In the examples we've seen so far in this book, only one-way data binding was used. The data displayed in the graphical components can't be edited to update the underlying DataSet. When two-way data binding is required, you can use the `AsDataView` query operator. `AsDataView` is an extension method provided by the `System.Data.DataTableExtensions` class. It creates a `DataView` instance for a collection of `DataRow`s returned by a LINQ to DataSet query.

`AsDataView` enables data binding scenarios for LINQ to DataSet. A `DataView` returned by `AsDataView` represents a LINQ to DataSet query itself and is not a view on top of the query.

The following are some benefits of the `DataView` returned by `AsDataView`:

- It's fully bindable
- It's fully updatable
- It reflects the changes happening in the underlying `DataTable`
- It keeps track of the LINQ to DataSet query's filtering and sorting expressions.

NOTE `AsDataView` can also create a `DataView` from a `DataTable`, providing a default view of that table.

But because the `AsDataView` query operator is available only for `DataTable` and `EnumerableRowCollection<T>`, it doesn't allow data binding scenarios for anonymous types. This means that it can't be used for joins and projections like the one we used in previous sections (see listing 14.12). Keep in mind that you can use `AsDataView` with a LINQ to DataSet query only if the query returns a collection of `DataRow`s.

Here is the typical scenario where `AsDataView` is used:

- 1 Data is loaded from a database into a `DataSet`.
- 2 The `DataSet` is queried using LINQ to `DataSet`, which allows all the powerful LINQ operations like joins, grouping, and sorting.
- 3 A `DataView` is created from the results thanks to `AsDataView`.
- 4 The `DataView` is bound to a `DataGridView`.
- 5 The data is edited in the `DataGridView`.
- 6 The original `DataSet` has been updated automatically by the `DataView` and can be used to update a database using a `DataAdapter` object.

Let's review sample code that demonstrates how to use `AsDataView` in another scenario. The method in listing 14.11 is used to bind a LINQ to `DataSet` `DataView` to a `DataGridView`.

Listing 14.11 Sample code that demonstrates the `AsDataView` query operator (FormMain.cs)

```
private void btnTypedDataView_Click(object sender, EventArgs e)
{
    LinqBooksDataSet dataSet = new LinqBooksDataSet();
    FillDataSetUsingLinqToSql2(dataSet);

    var books =
        from book in dataSet.Book.AsEnumerable()
        where book.Title.Length > 10
        orderby book.Title
        select book;

    DataView view = books.AsDataView();
    dataGridView1.DataSource = view;

    dataGridView2.DataSource = dataSet.Book;
}
```

Load a DataSet

Query a DataTable

Create a view on the query and bind it to the first DataGridView

Bind the Book DataTable to the second DataGridView

After the method has been executed, you can try to sort and update the data in the `DataGridViews`. You'll notice that the changes in either grid are reflected in the other one because the underlying `DataSet` is updated. You can also try to replace the title of a book with a shorter one to see that the filtering condition of the query is applied. Books with titles shorter than 10 characters aren't displayed in the first grid because the `DataView` applies the filter where `book.Title.Length > 10`.

You've now seen how to query DataSets with LINQ and how to use the query results. Before ending this chapter, we'd like to provide a reference for two major query operators that come with LINQ to DataSet—`Field` and `SetField`—and introduce a utility class that allows you to use set operators with LINQ to DataSet: `DataRowComparer`.

14.6 Using query operators with LINQ to DataSet

This section provides a quick reference concerning two query operators that are useful when querying DataSets. These query operators are provided as extension methods for the `DataRow` and `DataTable` classes. We'll first give you more information on the `Field` operator, which you've already seen. We'll also introduce the `SetField` operator, which is similar to `Field`, but for assignments. These operators will help you write richer LINQ queries against DataSets. We'll also review the `DataRowComparer` class provided with LINQ to DataSet.

14.6.1 `Field<T>` and `SetField<T>` operators for DataRows

Two query operators are provided as extension methods for `DataRow`: `Field` and `SetField`. These extension methods are provided by the `System.Data.DataRowExtensions` class.

`Field<T>`

We used `Field<T>` throughout this chapter, but it's good to review the purpose it serves. The `Field<T>` operator provides a way to get the value of a column within a `DataRow` without having to worry about the different representations of null values in DataSets and LINQ.

Let's look at the problem the `Field` operator solves. In DataSets, null values are represented using `System.DBNull.Value`. This is inconsistent with the way LINQ deals with null values. LINQ uses the support for nullable types introduced in the .NET Framework 2.0 release.

In our sample DataSet, the `Price` field accepts null values. DataSets don't support nullable types, so you can't write the following query:

```
from book in bookTable.AsEnumerable()  
where (Decimal?)book["Price"] > 10
```

The problem here is that when the `Price` field is null, `book["Price"]` returns `DBNull.Value`. `DBNull` is not convertible to `Nullable<Decimal>`, so the `where` clause throws an `InvalidCastException` if there's a row that has no value for the `Price` field.

In order to write safe code, you have to check whether the `Price` field is null prior to trying to access its value. Here's what the same query looks like with the check:

```
from book in bookTable.AsEnumerable()
where !book.IsNull("Price") && ((Decimal)book["Price"] > 10)
select book;
```

Here's how to rewrite the query using the `Field` operator:

```
from book in bookTable.AsEnumerable()
where book.Field<Decimal?>("Price") > 10
select book;
```

As you can see, the `Field` operator enables less verbose and less error prone queries because it automatically handles the conversion to nullable types.

SetField<T>

The `SetField` operator is provided as a companion to the `Field` operator to perform assignments that automatically handle null values.

Without the `SetField` operator, assigning a value to a `DataRow`'s field that can be null requires using a test such as the following to replace null with `DBNull.Value`:

```
book["Price"] = theValue ?? (Object)DBNull.Value;
```

NOTE In our code, we use the `??` operator. This is C#'s *null coalescing operator*. The test expression is equivalent to the following expression that uses the *ternary conditional operator*:

```
theValue == null ? (Object)DBNull.Value : theValue
```

With `SetField`, you can write the following:

```
book.SetField<Decimal?>("Price", theValue);
```

The benefit is not obvious, because the use of `SetField` doesn't really shorten the instruction, but `SetField` is here to mirror the `Field` operator.

Let's now introduce the `DataRowComparer` class.

14.6.2 Set operators and DataRow comparison with DataRowComparer

A number of set operators (`Distinct`, `Union`, `Intersect`, `Except`) exist in the standard query operators (see chapter 3). The problem with these operators is that they compare the equality of source elements by calling the `GetHashCode` and `Equals` methods on each collection of elements. In the case of `DataRows`, this performs a

reference comparison, which is generally not the ideal behavior for set operations over tabular data. For set operations, you usually want to determine whether the element values are equal and not the element references. Therefore, the `DataRowComparer` class has been added to LINQ to DataSet. This class can be used to compare row values.

The `DataRowComparer` class contains a value comparison implementation for `DataRow`, so this class can be used for set operations such as `Distinct`.

NOTE The `DataRowComparer` class can't be directly instantiated. Instead, the Default property of the class provides a singleton instance ready for use.

The `Equals` method of the `DataRowComparer` class is called when two `DataRow` objects should be compared. This method returns true if the ordered set of column values in both `DataRow` objects it receives as input parameters are equal; otherwise, it returns false.

Here's a sample use of the `Intersect` query operator:

```
private void btnIntersect_Click(object sender, EventArgs e)
{
    LinqBooksDataSet dataSet = new LinqBooksDataSet();
    FillDataSetUsingLinqToSql2(dataSet);

    var query1 =
        from book in dataSet.Book.AsEnumerable()
        where book.Price < 30
        select book;
    var query2 =
        from book in dataSet.Book.AsEnumerable()
        where book.PageCount > 100
        select book;
    var books1 = new LinqBooksDataSet.BookDataTable();
    query1.CopyToDataTable(books1, LoadOption.PreserveChanges);
    var books2 = new LinqBooksDataSet.BookDataTable();
    query2.CopyToDataTable(books2, LoadOption.PreserveChanges);

    IEqualityComparer<LinqBooksDataSet.BookRow> comparer =
        new DataRowComparer<LinqBooksDataSet.BookRow>();
    var books = books1.AsEnumerable()
        .Intersect(books2.AsEnumerable(), comparer)
        .Select(book => new { book.Title, book.Price, book.PageCount });

    dataGridView2.DataSource = books.ToList();
}
```

Load a DataSet

Create two tables

Find the intersection of the two tables

Display books costing less than 30 that have more than 100 pages

For this sample to work with typed DataSets, we created a custom generic comparer. Here's how it's implemented:

```
public class DataRowComparer<TDataRow> :  
    IEqualityComparer<TDataRow>  
where TDataRow: DataRow  
{  
    public bool Equals(TDataRow x, TDataRow y)  
    {  
        return DataRowComparer.Default.Equals(x, y);  
    }  
  
    public int GetHashCode(TDataRow obj)  
    {  
        return DataRowComparer.Default.GetHashCode(obj);  
    }  
}
```

With `DataRowComparer`, you can use the set operators in your LINQ to DataSet queries and have them behave as expected.

14.7 Summary

LINQ to DataSet is a part of the LINQ toolset that joins LINQ to Objects, LINQ to XML, and LINQ to SQL to enable language-integrated queries in a wide range of scenarios. In this chapter, you've seen how LINQ to DataSet enables scenarios that involve DataSets and LINQ queries. This LINQ flavor is important because DataSets are commonly used in .NET applications.

If you already use DataSets, you can now to improve your existing code with rich queries. If you don't already use DataSets, you may reconsider them in light of their new querying capabilities.

Again, one big advantage of LINQ is that it enables a consistent querying infrastructure for a variety of data structures. If you already know LINQ to Objects or the standard query operators, you'll be able to use LINQ to DataSet easily.

LINQ in Action

Fabrice Marguerie • Steve Eichert • Jim Wooley

NET applications are object-oriented, but the data is not. That's the situation when you're using a relational database, XML, and many other data stores, and for each you need a separate programming solution. Microsoft's Language INtegrated Query, known as LINQ, is a set of .NET Framework and language extensions that offers a single, simple way to query data of any form directly from C# 3 and VB.NET 9. On top of that, your persistence code gets the same compile-time syntax checking, static typing and IntelliSense available to the rest of your code.

Written for C# and VB developers of all levels, **LINQ in Action** ramps up quickly from zero knowledge at first to a substantial depth at the end. In it, you'll explore the key language features like lambda expressions, extension methods, and anonymous data types that make LINQ possible. Following a running example, the book walks you through core techniques to query objects, relational databases, and XML. You'll master the Standard Query Operators along with the instantly-familiar SQL-like syntax of LINQ's query expressions. You'll also learn to build custom LINQ solutions such as the book's clever "LINQ to Amazon."

What's Inside

- Fully tested against the final version of .NET 3.5
- All code examples in both C# 3 and VB.NET 9
- LINQ to Objects, LINQ to SQL, LINQ to XML, and more
- How to do domain-specific LINQ customization

Fabrice Marguerie is a software architect and developer based in Paris, France. A C# MVP, Fabrice has worked with LINQ from the first prototypes. **Steve Eichert** is an architect with Algorithmics, Inc. based in Philadelphia, PA. **Jim Wooley** is a VB.NET MVP, INETA Membership Mentor for Georgia, and frequent speaker at user events.

For more information, code samples, and to purchase an ebook visit manning.com/LINQinAction

"It's like they threw a party for LINQ and everyone who's anyone showed up."

—FROM THE FOREWORD BY
Matt Warren
Principal Architect, Microsoft

"Great if you want to fully grok LINQ."

—Javier Lozano, lozanotek.com

"Very useful—both straightforward and pragmatic."

—Bruno Boucard
Microsoft France

"Teaches you to think in LINQ. Wonderfully complete."

—Jon Skeet, C# MVP and
author of *C# in Depth*

"Covers LINQ, inside & out."

—Mohammad Azam
University of Houston

"A great guide to all things LINQ!"

—Tomas Restrepo, devdeo.lt