Microsoft

developer

// Step by step

Microsoft ADO.NET Entity Framework

Intermediate

John Paul Mueller

// Step by step

Your hands-on guide to Entity Framework fundamentals

Expand your expertise—and teach yourself the fundamentals of the Microsoft ADO.NET Entity Framework 5. If you have previous programming experience but are new to the Entity Framework, this tutorial delivers the step-by-step guidance and coding exercises you need to master core topics and techniques.

Discover how to:

- Access data in a managed way—using minimal code
- Apply three workflows supported by the Entity Framework
- Perform essential tasks with full automation in place
- Manipulate data with both LINQ and Entity SQL
- Create examples that rely on Table-Valued Functions
- Determine the remedies for Entity-specific exceptions
- Explore the use of optimistic and pessimistic concurrency
- Define mappings between your applications and data sources

About the Author

John Paul Mueller, a technical editor and freelance writer, has covered topics ranging from database management to heads-down programming, and from networking to artificial intelligence. He is the author of *Start Here! Learn Microsoft Visual C# 2010*.

Practice Files + Code

Available at:
http://aka.ms/ADONETEFSbS/files

Companion eBook

See the instruction page at the back of the book

microsoft.com/mspress

ISBN: 978-0-7356-6416-6

9 780735 664166 90000

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Programming/Microsoft .NET

Microsoft

Microsoft

Microsoft ADO.NET Entity Framework Step by Step

John Paul Mueller

ISBN: 978-0-735-66416-6

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

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Christopher Hearse

Editorial Production: Zyg Group, LLC

Technical Reviewer: Russ Mullen

Indexer: Zyg Group, LLC

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

Illustrator: Rebecca Demarest

This book is dedicated to Kevin Smith, a good friend who's helped us realize some of our most special dreams. He's always helped us help ourselves—an outstanding gift that's exceptionally rare in this world.

—John Paul Mueller

Contents at a glance

Contents

Introduction

Gaining access to data in a managed way without a lot of coding—that's a tall order! The Entity Framework fulfills this promise and far more. Each version of the Entity Framework is more capable than the last. The latest version, Entity Framework version 5, provides you with access to far more database features with less work than ever before, and *Microsoft ADO.NET Entity Framework Step by Step* is your gateway to finding just how to use these phenomenal new features. In this book, you get hands-on practice with all the latest functionality that the Entity Framework provides. By the time you finish, you'll be ready to tackle some of the most difficult database management tasks without the heavy-duty coding that past efforts required.

Fortunately, this book doesn't get so immersed in high-end features that it forgets to tell you how to get started. Unlike a lot of tomes on the topic, this book starts simply and helps you gain a good foothold in understanding just why the Entity Framework is such an amazing addition to the your developer toolbox. You'll see examples where the automation does just about everything for you with little coding required, and yet you obtain professional-looking results. In fact, that's what you're buying with the Entity Framework—a reliable means of creating code quickly and successfully without the problems that would ensue if you tried to create the same code completely by hand. The book's 44 examples help you gain experience using the Entity Framework in a hands-on environment where you actually create code, rather than just reading about what might work.

Of course, you do eventually delve into higher-end topics. You'll find an entire chapter on one of the most requested features, Table-Valued Functions (TVFs). Access to this feature alone makes the upgrade to Entity Framework 5 a significant one. You'll also discover how to handle performance problems and perform low-end tasks such as using inheritance when creating a model. In short, by the time you finish this book, you will have the experience required to handle every common task that developers need to know how to perform.

Who should read this book

Anyone who creates database applications using ADO.NET and is tired of writing reams of code will definitely benefit from reading this book. What you should ask yourself is whether you want to become more productive while writing code that is both more

reliable and better able to interact with the database. Although the coding examples are written in C#, several Microsoft Visual Basic developers tested this book during the writing process and found that they could follow the examples quite well. All you really need is a desire to write database applications more quickly and with less fuss.

Assumptions

To use this book successfully, you need a good knowledge of database programming concepts using a technology such as ADO.NET. Although every attempt is made to explain basic (and essential) topics, a knowledge of working with databases using the .NET Framework will make working through the examples significantly easier.

You also need to know how to write applications using the C# programming language. All of the examples are written using C#, and there isn't any attempt to explain how the language elements work. If you don't have the required C# knowledge, you should consider getting John Sharp's *Microsoft Visual C# 2010 Step by Step* (Microsoft Press, 2010).

Some of the examples also require some knowledge of Transact-Structured Query Language (T-SQL). Again, there are plenty of comments provided with the various scripts, but there isn't a lot of additional information provided about language elements. The book assumes that you know how basic SQL queries work.

Who should not read this book

This book is most definitely not aimed at the complete novice. You must know a little about both SQL and ADO.NET to work with the book successfully. In addition, you must know the C# programming language fairly well. The examples in the book focus a little more on enterprise developers, but hobbyists should be able to follow the examples without problem. If you're looking for a high-end book with lots of low-end programming examples and no hands-on techniques, this is most definitely not the book for you. This book is all about getting people started using the Entity Framework in a meaningful way to perform most common tasks, which means it uses several different techniques to convey information so that a majority of readers can understand and use the material presented.

Organization of this book

This book is organized into five parts. Each part is designed to demonstrate a particular facet of the Entity Framework, with an emphasis on the functionality provided by version 5. Here is a brief overview of the book parts (each part introduction has more detailed information about the content of the chapters in that part):

- **Part I: Introducing the Entity Framework** This part of the book introduces you to the Entity Framework version 5. You'll discover what is new in this version of the Entity Framework and also basic concepts such as the parts of a model. Unlike many other texts, this part also tells you about the three workflows available when working with the Entity Framework: model first, database first, and code first. Every chapter includes coding examples that emphasize the basics so that you can see precisely how the Entity Framework works at a basic level.
- **Part II: Completing basic tasks** Once you have a basic understanding of what the Entity Framework does and why you'd want to use it, it's time to see how to perform basic Create, Review, Update, and Delete (CRUD) operations. This part of the book provides an essential discussion of how to perform essential tasks with full automation in place. It's the part of the book you want to read to emphasize speed of development over flexibility in accessing database functionality.
- **Part III: Manipulating data using the Entity Framework** Most applications require more than a display of raw database data and simple CRUD operations. This part of the book takes the next step in your journey of actually controlling how the data appears and precisely what data is retrieved from the database. You discover two client-side techniques for manipulating data (Language Integrated Query [LINQ] and Entity Structured Query Language [Entity SQL]). In addition, you see how to use server-based techniques that include stored procedures, views, and TVFs.
- **Part IV: Overcoming entity errors** It's nearly impossible to create an application that is free from error. In fact, smart developers know that it is impossible because you really never have full control over absolutely all of the code that goes into your application. This part of the book discusses three realms of error: exceptions, concurrency issues, and performance problems.

- **Part V: Advanced management techniques** This is the low-level-coding part of the book. This is where you learn how to create custom entities and use inheritance as a tool to create more robust models. You also discover techniques for mapping various kinds of data to the Entity Framework, even when the Entity Framework normally doesn't support the data type. The key thing to remember about this part is that you discover manual methods for modifying how the automation works.

Finding your best starting point in this book

The different sections of *Microsoft® ADO.NET Entity Framework Step by Step* cover a wide range of technologies associated with the Entity Framework. Depending on your needs and your existing understanding of Microsoft data tools, you may wish to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to the Entity Framework	Begin with Chapter 1, "Getting to know the Entity Framework," and move through Chapter 13, "Handling performance problems." Skip the last part of the book until you have gained some experience using the automation that the Entity Framework provides.
Familiar with earlier releases of the Entity Framework	Read through Chapter 1 and Chapter 3, "Choosing a workflow," carefully. Chapter 3 is especially important because it helps you understand the new workflows. Work through Parts III, IV, and V as needed to update your knowledge.
Interested in learning advanced Entity Framework techniques	Move directly to Part V of the book. The first four parts of this book are designed to help you learn about the Entity Framework and interact successfully with the automation it provides.
Interested in using the existing database infrastructure of your organization	Read Parts I and II to ensure you understand the basics of how the Entity Framework works, and then skip to Chapter 8, "Interaction with stored procedures," Chapter 9, "Interaction with views," and Chapter 10, "Interaction with table-valued functions."

Every chapter in this book contains at least one hands-on example (and usually more). The only way you'll actually gain a full understanding of the Entity Framework is to download the sample code and then work through the hands-on examples. Each of these procedures demonstrates an important element of the Entity Framework.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow.

Note Note boxed elements tell you about additional information that will prove useful in working with the Entity Framework. Notes normally include text about techniques used to create examples or the sources of information used in creating the chapter's content.

Tip Tip boxed elements provide additional information that will enhance your productivity, make it easier to perform tasks, or help you locate additional sources of information. Most tips provide helpful information that you don't need to know in order to use the book, but the information will prove helpful later as you work with real-world code.

Warning Warning boxed elements describe potentially dangerous situations where performing an act could result in damage to your application, the data it manages, or the user environment (such as the need to keep certain types of information secure). Pay special attention to warning elements because they'll save you time and effort.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Sidebars contain useful information that isn't part of the main flow of discussion in a chapter. These elements always have a title that tells you about the topic of discussion. You can safely skip sidebars if desired or simply read them later. Sidebars always provide you with helpful real-world resource information that will help you as you create or manage applications.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

- A vertical bar between two or more menu items (for example, File | Close) means that you should select the first menu or menu item, then the next, and so on.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- A copy of Microsoft Windows that will work with Microsoft Visual Studio 2012, which can include Windows 7 SP1 (x86 and x64), Windows 8 (x86 and x64), Windows Server 2008 R2 SP1 (x64), or Windows Server 2012 (x64).
- A copy of Visual Studio 2012 Professional or better. This book won't work well with Visual Studio 2012 Express Edition. In fact, many of the examples won't work at all, even if you use the downloaded source code.
- A copy of Microsoft SQL Server 2012 Express Edition with SQL Server Management Studio 2012 Express or higher (included with Visual Studio). You can also use the full-fledged version of SQL Server 2012.

Your computer must also meet these minimum requirements (although higher ratings are always recommended):

- 1.6 GHz or faster processor
- 1 GB of RAM (1.5 GB if running on a virtual machine)
- 10 GB of available hard disk space
- 600 MB of available hard disk space
- 5400 RPM hard drive
- DirectX 9–capable video card running at 1024×768 or higher display resolution
- DVD drive

Your computer must also have access to an Internet connection to download software or chapter examples.

Note Many of the tasks in this book require that you have local administrator rights. Newer versions of Windows include stricter security that requires you to have additional rights to perform tasks such as creating copies of database files.

Code samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

http://aka.ms/ADONETEFSbS/files

Follow the instructions to download the zip file.

Note In addition to the code samples, your system should have Visual Studio 2012 Professional (or better) and SQL Server 2012 Express Edition (or better) installed. The exercises will include instructions for working with SQL Server 2012. In most cases, the exercises rely on Server Explorer to make it easy to perform all tasks from the Visual Studio Integrated Development Environment (IDE).

Installing the code samples

All you need to do to install the code samples is download them and unzip the archive to a folder on your hard drive. The complete source code file will include all of the databases used in the book. Simply attach these databases to your copy of SQL Server or open them in Visual Studio by right-clicking Data Connections in Server Explorer and choosing Add Connection. Use the Microsoft SQL Server Database Connection option when creating the connection. If you encounter problems installing the code samples, please contact me at *John@JohnMuellerBooks.com*. You can also find answers to common questions for this book on my blog, at *http://blog.johnmuellerbooks.com/categories/263/entity-framework-development-step-by-step.aspx*.

Using the code samples

The downloaded source code includes one folder for each chapter in the book. Simply open the chapter folder and then the example folder for the example you want to work with in the book. The downloaded source contains the completed source code so that you can see precisely how your example should look. If you want to work through the examples from scratch, the book contains complete instructions for developing them.

The downloaded source code also contains a Databases folder that contains all of the databases for the book. Simply create a connection to the database you need to use. The example will tell you which database is required. If you desire, the exercises also tell you how to create the databases from scratch so that you can use whatever setup you like.

Acknowledgments

Thanks to my wife, Rebecca, for working with me to get this book completed. I really don't know what I would have done without her help in researching and compiling some of the information that appears in this book. She also did a fine job of proofreading my rough draft. Rebecca keeps the house running while I'm buried in work.

Russ Mullen deserves thanks for his technical edit of this book. He greatly added to the accuracy and depth of the material you see here. Russ is always providing me with great URLs for new products and ideas. However, it's the testing Russ does that helps most. He's the sanity check for my work. Russ also has different computer equipment from mine, so he's able to point out flaws that I might not otherwise notice.

Matt Wagner, my agent, deserves credit for helping me get the contract in the first place and taking care of all the details that most authors don't really consider. I always appreciate his assistance. It's good to know that someone wants to help.

A number of people read all or part of this book to help me refine the approach, test the coding examples, and generally provide input that all readers wish they could have. These unpaid volunteers helped in ways too numerous to mention here. I especially appreciate the efforts of Eva Beattie and Glenn Russell, who provided general input, read the entire book, and selflessly devoted themselves to this project.

Finally, I would like to thank my editor, Russell Jones; Christopher Hearse; Damon Larson; and the rest of the editorial and production staff at O'Reilly for their assistance in bringing this book to print. It's always nice to work with such a great group of professionals.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/ADONETEFSbS/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

PART I

Introducing the Entity Framework

Creating a database can be difficult. A database models information in the real world using a collection of tables, indexes, views, and other items. In other words, a database is an abstraction of the real-world information that it's supposed to represent. When a developer is tasked with creating an application that relies on the data within a database, the developer must create a second level of abstraction because the application won't see the data in precisely the same way that the database does. Defining this second level of abstraction is even harder than creating the original database, because it requires interpreting the real world through an abstraction. In order to define a realistic presentation of the data in the database—one that precisely represents the real world—a developer needs help. That's what the Entity Framework does. It provides help to a developer in the form of a modeling methodology that eases the amount of work the developer must perform to create a realistic presentation. To make things even easier, the Entity Framework relies on a graphical presentation of the data so that the developer can literally see the relationships between the various tables and other database items.

Even though the concept of the Entity Framework is straightforward, you need to know more about it before you can simply use it to create a connection between the database and your application. Working with models is definitely easier than working with hand-coded connections. However, you still need to have a good understanding of how those models work and the various ways you can interact with them. The purpose of the three chapters in this part is to introduce you to the Entity Framework concepts. You'll use this information to build a picture of how the Entity Framework performs its task so that you can perform more complex operations with the Entity Framework later in the book.

CHAPTER 1

Getting to know the Entity Framework

After completing the chapter, you'll be able to

- Define what an entity is and why it's important.
- Specify the major elements of the Entity Framework.
- List and describe the files used to store Entity Framework information.
- Create a simple Entity Framework example.

When an architect wants to design a real-world building by creating a blueprint, one of the tools used to ensure the blueprint is accurate is a *model*. Often you see a model of the building as part of the presentation for that building. Models are helpful because they help others visualize the ideas that reside in the architect's head. In addition, the models help the architect decide whether the plan is realistic. Likewise, software developers can rely on models as a means of understanding a software design, determining whether that design is realistic, and conveying that design to others. The Entity Framework provides the means to create various kinds of models that a developer can interact with in a number of ways. As with the architect's model, the Entity Framework uses a graphical interface to make information about the underlying database structure easier to understand and modify.

The Entity Framework is actually a Microsoft ActiveX Data Object .NET (ADO.NET) technology extension. When you create the model of the database, you also make it possible for the Integrated Development Environment (IDE) to automatically create some of the code required to make the connection between an application and the database real. Because of the way ADO.NET and the Entity Framework interact, it's possible to create extremely complex designs and then use those designs directly from your code in a way that the developer will understand. There isn't any need to translate between the levels of abstraction—the Entity Framework performs that task for you.

Before you can begin using the Entity Framework to perform useful work, however, you need to know a little more about it. For one thing, you need to know why it's called an Entity Framework. It's also important to know how the various models work and how they're stored on your system, should you ever need to access them directly. The following sections provide this information and more about the Entity Framework. You'll then use the knowledge you've gained to create a very simple example. This example will help you better understand what the Entity Framework can do because you'll actually use it to interact with a simple database.

Defining an entity

An *entity* is the data associated with a particular object when considered from the perspective of a particular application. For example, a customer object will include a customer's name, address, telephone number, company name, and so on. The actual customer object may have more data than this associated with it, but from the perspective of this particular application, the customer object is complete by knowing these facts. If you want to understand this from the traditional perspective of a database administrator, the entity would be a single row in a view that contains all of the related information for the customer. It includes everything that the database physically stores in separate tables about that particular client. When thinking about entities, you need to consider these views of the data:

- **Physical** The tables, keys, indexes, views, and other constructions that hold and describe the data associated with a real-world object such as a customer. All of these elements are optimized to make it easier for the Database Management System (DBMS) to store and manipulate the data efficiently and reliably, without error. As such, a single customer data entry can appear in multiple tables and require the use of multiple keys to create a cohesive view of that customer. The physical storage of the data is efficient for the DBMS, but difficult for the developer to understand.

- **Logical** The combined elements required to define the data used with a single object, such as a client. From a database perspective, the logical view of the data is often encapsulated in a *view*. The view combines the data found within tables using keys and other database elements that describe the relationships and order required to re-create the customer successfully. Even so, the logical view of a database is still somewhat abstract and could cause problems for the developer, not to mention require a lot of code to manage successfully. ADO.NET does reduce the amount of coding the developer performs through the use of built-in objects, but the developer must still understand the underlying physical construction of that data.

- **Conceptual** The real-world view of the data as it applies to the object. When you view a customer, you see attributes that define the customer and remember items that describe the customer, such as the customer's name. A conceptual view of the data presents information in this understandable manner—as objects where the focus is on the data, not on the structure of the underlying database.

When you want to think about customers as a group, you work with entities. Each entity is a single customer, and the customers as a group are entities as well. In order to visualize the data that comprises a customer, the Entity Framework relies on models. These models help the developer conceptualize the entities. In addition, the Entity Framework stores these models in XML format for use in automatically generating code to create objects based on the models. Working with objects makes life easier for the developer.

Note You may be tempted to think of the Entity Framework as a technology that only applies to Microsoft SQL Server and other relational databases. The Entity Framework is a full solution that works with any data source, even flat-file and hierarchical databases. For the sake of making the discussion clear, this book will rely upon SQL Server for the examples, but you should know that using SQL Server is only a convenience, and you can use the Entity Framework for any data source your application needs to work with. In addition, you can mix and match data sources as needed within a single application.

In times past, developers needed to consider the physical (tables), logical (views), and conceptual (data model) perspectives of data stored in a database. A developer had to know precisely which table stored a particular piece of data, how that table was related to other tables in the database, and how to relate the data in such a way as to create a complete picture of a particular entity. The developer then wrote code to make the connectivity between the application and the database work. The Entity Framework reduces the need to perform such tasks. A developer focuses on the entity, not the underlying physical or logical structure of the database that contains the data. As a result, the developer is more productive. Working with entities also makes the data easier to explain to others.

An entity contains *properties*. Just as objects are described by the properties they contain, entities contain individual properties that describe each data element. A customer's last name would be a property of a customer entity. Just as classes have configurable getters and setters, so do properties in the Entity Framework. Every entity has a special property called the key property. The *key property* uniquely defines the entity in some way. An entity can have more than one key property, but it always has at least one. An entity can also group multiple properties together to create a complex type that mimics the use of user-defined types with standard classes.

Note It's important to remember that properties can contain either simple or complex data. Simple data is of a type defined by the .NET Framework, such as *Int32*. Complex data is more akin to a user-defined type and consists of multiple base types within a structure-like context.

It's possible to create a relationship between two entities through an association. For example, you might create an association between a customer entity and the order entities associated with the customer. The association type defines the specifics of the association. In some respects, an association is similar to a database-level join. One or more properties in each entity, called *association endpoints*, define the relationship between the two entities. The properties can define both single and multicolumn connections between the two entities. The multiplicity of the association endpoints determines whether the association is one-to-one, one-to-many, or some other combination. The association

is bidirectional, so entities have full access to each other. In addition, an entity association can exist even when the target data lacks any form of database-level join specification. All of the association instances used to define an association type make up what is called the *association set*.

In order to allow one entity to view the data provided by an associated entity, the entities have a navigation property. For example, a customer entity that's associated with multiple order entities will have a navigation property that allows each order to know that it's associated with that customer. Likewise, each order will have a navigation property that allows each customer entity to see all of the orders associated with it. The use of navigation properties allows your code to create a view of the entities from the perspective of a particular entity. When working with a customer entity, the application can gain access to all of the orders submitted by that customer. In some respects, this feature works much like a foreign key does in a database, but it's easier to work with and faster to implement.

Some entities derive from other entities. For example, a customer can create an order. However, the order will eventually have a state that creates other entities, such as a past-due order entity or a delivered-order entity. These derived entities exist in the same container as the order entity as part of an entity set. You can view the relationship between entities and derived entities as being similar to a database and its views. The database contains all of the data, but a view looks at the data in a particular way. In the same way, a derived entity would help you create applications that view a particular entity type within the set of entities.

The final piece of information you need to know for now about entities concerns the *entity container*. In order to provide a convenient means to hold all of the entity information together, the Entity Framework employs the entity container. Before you can interact with an entity, your application creates an entity container instance. This instance is known as the *context*. Your application accesses the entities within a particular context.

Understanding the Entity Framework elements

The Entity Framework relies on XML files to perform its work. These files perform three tasks: defining the conceptual model, defining the storage model, and creating mappings between the models and the physical database. Even though the Entity Framework does a lot of the work for you, it's still important to understand how these elements work together to create a better environment in which to write applications.

Note This chapter discusses the idea of models generically. However, it's important to realize that the Entity Framework lets you interact with the database using one of three techniques:

- **Database first** The Entity Framework creates classes that reflect an existing database design.
- **Design first** You define a model of the database that the Entity Framework then creates on the database server.
- **Code first** You create an application, and the Entity Framework defines a model that it then uses to create the database on the database server.

In all three cases, the Entity Framework eventually creates a model that follows the standards described in this chapter. You'll learn more about the methods of working with the Entity Framework in Chapter 3, "Choosing a workflow." For now, the important consideration is the model itself.

Now that you a have a little idea of what constitutes the Entity Framework elements, it's time to discuss them in greater detail. In this case, we're looking at the logical structure of the Entity Framework. The physical structure (the XML files and their content) is discussed in the "Introducing the Entity Framework files" section of the chapter. The following sections discuss the conceptual model, storage model, and model mappings.

Considering the conceptual model

The conceptual model is the part of the Entity Framework that developers interact with most. This model defines how the database looks from the application's perspective. Of course, the application view must somehow match the physical realities of the underlying database, but there are many ways in which this happens. For example, a C# application will use an *INT32* value, rather than the Structured Query Language (SQL) *int* type. The conceptual model will refer to the data type as *INT32*, but the reality is that the database itself stores the data as an *int*.

The conceptual model is also used to create the classes used to interact with the database. The Entity Framework manages the conceptual model. As you make changes to the conceptual model, the changes are reflected in both the classes that the Entity Framework creates for your application and in the structure of the database. In addition, the Entity Framework automatically tracks changes to the database design and incorporates them into your implementation classes. As a result, your application can always access the data and functionality included with the target database.

Note It's important to realize that changes to the database design can occur at several levels. The two most common levels are from the developer, when making changes to the database model to accommodate application requirements; and from the Database Administrator (DBA), to accommodate enterprise-wide changes to the database as needed to efficiently and reliably store information. No matter how a change occurs, the database structure is ultimately affected, at which point the Entity Framework detects the change and updates the application using the data.

A conceptual model also incorporates the concept of a namespace, just as your applications do. An Entity Framework namespace performs the same functions as the namespace in your application. For example, it helps define entities with the same name as unique features. Using namespaces also helps group like entities together. For example, everything related to a customer can appear in the same namespace, making it easier to interact with the customer in every way needed.

At the heart of the conceptual model are the entity and association definitions used to create the view of the database. Each entity definition includes the information described in the "Defining an entity" section earlier in this chapter. When you use the designer to interact with the database model, what you're really doing is modifying the XML entries that create and define each of these entity definitions. The XML entries are stored on disk and used to re-create the graphic appearance of the model when you reopen the project.

Considering the storage model

The storage model is the part of the Entity Framework that defines how the database looks from the database manager's perspective. However, this model provides this view within Microsoft Visual Studio, and it provides support for the conceptual model. This model is often called the logical model because it provides a logical view of the database that ultimately translates into the physical database (see the "Defining an entity" section earlier in this chapter for a description of the various database views).

As with the conceptual model, the storage model consists of entity and association definitions. However, these definitions reflect the logical appearance of the actual database, rather than the presentation of the conceptual model within the application. In addition to the entity and association definitions, the storage model includes actual database data such as commands used to query the information within the database. You'll also find stored procedures in this model. All of this additional information is used by ADO.NET to create connection and command objects automatically, so that you don't have to hand-code the information as part of your application.

Considering the model mappings

At this point, you know that there are two models used with the Entity Framework—the conceptual model presents the application view of the database and the storage model presents the logical database manager view of the database. These two models are necessarily different. If they were the

same, you wouldn't need two models. The need for two models is also easy to understand once you consider that the application's use of the database is always going to differ from the database manager's goals of storing the data efficiency and reliably. In order to make the two models work together, the Entity Framework requires model mapping—a third element that describes how something in the conceptual model translates to the storage model, and vice versa.

The overall goal of the model-mapping part of the Entity Framework is to create a definition of how the entities, properties, and associations in the conceptual model translate to elements within the storage model. This mapping makes it possible for the application to create a connection to the database, modify its structure, manage data, and perform other tasks with a minimum of manually written code. Most of the code used to interact with the database is automatically generated for the developer using the combination of the conceptual model, storage model, and this mapping layer.

Introducing the Entity Framework files

As previously mentioned, all of the files used with the Entity Framework rely on XML. The use of XML makes the files portable and easy to use with other applications. You can also view the content of these files and reasonably expect to understand much of what they contain. However, each of the Entity Framework elements uses a different XML file with a different file extension and a different language inside.

After you create a new application that relies on the Entity Framework and define the required database models, you can find the resulting files in the main folder of the project. When working with Visual Studio 2012, you'll find a single Entity Data Model XML (.EDMX) file. However, when working with older versions of Visual Studio, you may find individual files for each of the Entity Framework elements.

Providing a complete tutorial on each of these files is outside the scope of this book. The following sections provide a useful overview of the files, which you can use for further study.

Viewing the Conceptual Schema Definition Language file

The Conceptual Schema Definition Language (.CSDL) file contains the XML required to build the conceptual model of the database content as viewed by the application. You see this content in graphical format when working with Visual Studio. To see it in plain-text form, locate the .CSDL or .EDMX file for your application in the project folder. Right-click this file in Microsoft Windows Explorer and choose Open With from the context menu. Locate Notepad or some other suitable text editor in the Open With dialog box, clear any option that says that this program will become the default program for opening this file, and click OK. You'll see the XML for the conceptual model for the application. Following is the XML for the sample application that appears later in the chapter (some <*Schema*> attributes are removed to make the listing easier to read).

Note When using Visual Studio design tools to create the .CSDL, Store Schema Definition Language (.SSDL), and Mapping Specification Language (.MSL) files, all three are stored in a single .EDMX file, rather than in separate files. Whether the data appears in a single file or within multiple files, it's always stored as XML. An .EDMX file also contains some designer information not found in the separate files. You can safely ignore the designer information when viewing the .EDMX file in order to understand how the conceptual model, storage model, and model mapping interact.

```
<!-- CSDL content -->
<edmx:ConceptualModels>
  <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"...>
    <EntityContainer Name="Model1Container" annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Customers" EntityType="Model1.Customer" />
    </EntityContainer>
    <EntityType Name="Customer">
      <Key>
        <PropertyRef Name="CustomerID" />
      </Key>
      <Property Type="Int32" Name="CustomerID" Nullable="false"
                annotation:StoreGeneratedPattern="Identity" />
      <Property Type="String" Name="FirstName" Nullable="false" />
      <Property Type="String" Name="LastName" Nullable="false" />
      <Property Type="String" Name="AddressLine1" Nullable="false" />
      <Property Type="String" Name="AddressLine2" Nullable="false" />
      <Property Type="String" Name="City" Nullable="false" />
      <Property Type="String" Name="State_Province" Nullable="false" />
      <Property Type="String" Name="ZIP_Postal_Code" Nullable="false" />
      <Property Type="String" Name="Region_Country" Nullable="false" />
    </EntityType>
  </Schema>
</edmx:ConceptualModels>
```

The XML makes it easier to understand the preceding discussion of how an *Entity* object works. Notice that the XML describes an entity container, used to hold all of the entities for this particular model. Within that container is a single *EntityType* named *Customer*. As with all *Entity* objects, this one has a *Key* property named *CustomerID* that gives the *Entity* a unique value. In addition, there are a number of properties associated with this *Entity*, such as *FirstName*. You'll see how the properties work later in the chapter. Of course, an *Entity* can have other elements associated with it, and you'll see them at work later in the book.

Look at the individual <*Property*> entries. Each one includes a .NET type. In this case, the types are limited to *Int32* and *String*, but you have access to a number of other types. You can see the primitive data types supported by the Entity Framework at *http://msdn.microsoft.com/library/ee382832.aspx*.

Viewing the Store Schema Definition Language file

The .SSDL file contains the XML required to define the storage model of the database content as viewed by the database manager. As with the conceptual model, you see the database described in terms of the entities required to create it. The entries rely on SQL data types, rather than .NET data types. Here's an example of the XML used to create a storage model for the example that appears later in the chapter (the <*Schema*> has been shortened to make the text easier to read):

```
  <!-- SSDL content -->
  <edmx:StorageModels>
  <Schema Namespace="Model1.Store" Alias="Self"...>
<EntityContainer Name="Model1StoreContainer">
  <EntitySet Name="Customers" EntityType="Model1.Store.Customers" store:Type="Tables"
             Schema="dbo" />
</EntityContainer>
<EntityType Name="Customers">
  <Key>
    <PropertyRef Name="CustomerID" />
  </Key>
  <Property Name="CustomerID" Type="int" StoreGeneratedPattern="Identity"
            Nullable="false" />
  <Property Name="FirstName" Type="nvarchar(max)" Nullable="false" />
  <Property Name="LastName" Type="nvarchar(max)" Nullable="false" />
  <Property Name="AddressLine1" Type="nvarchar(max)" Nullable="false" />
  <Property Name="AddressLine2" Type="nvarchar(max)" Nullable="false" />
  <Property Name="City" Type="nvarchar(max)" Nullable="false" />
  <Property Name="State_Province" Type="nvarchar(max)" Nullable="false" />
  <Property Name="ZIP_Postal_Code" Type="nvarchar(max)" Nullable="false" />
  <Property Name="Region_Country" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

Viewing the Mapping Specification Language file

The .MSL file creates a relationship between the .CSDL and .SSDL files. The mapping serves to define how the application view and the database manager view reflect the same database, but from differing perspectives. For example, the model mapping defines which conceptual model property translates into a particular storage model property. Here's the model-mapping content for the example that appears later in the chapter:

```
<!-- C-S mapping content -->
<edmx:Mappings>
<Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
<EntityContainerMapping StorageEntityContainer="Model1StoreContainer"
                        CdmEntityContainer="Model1Container">
  <EntitySetMapping Name="Customers">
    <EntityTypeMapping TypeName="IsTypeOf(Model1.Customer)">
```

```
      <MappingFragment StoreEntitySet="Customers">
        <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
        <ScalarProperty Name="FirstName" ColumnName="FirstName" />
        <ScalarProperty Name="LastName" ColumnName="LastName" />
        <ScalarProperty Name="AddressLine1" ColumnName="AddressLine1" />
        <ScalarProperty Name="AddressLine2" ColumnName="AddressLine2" />
        <ScalarProperty Name="City" ColumnName="City" />
        <ScalarProperty Name="State_Province" ColumnName="State_Province" />
        <ScalarProperty Name="ZIP_Postal_Code" ColumnName="ZIP_Postal_Code" />
        <ScalarProperty Name="Region_Country" ColumnName="Region_Country" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
</EntityContainerMapping>
</Mapping>
</edmx:Mappings>
```

Developing a simple Entity Framework example

The best way to begin learning about the Entity Framework is to use it. This example won't do anything too spectacular. In fact, it's downright mundane, but it does reflect a process that many developers use to experiment with the Entity Framework. In this case, you'll use the model-first technique to create an example application. Remember that in the model-first approach, you begin by creating a model that's then added to the database, rather than relying on an existing database to define the model. The model-first technique has the advantage of allowing you to create and manipulate a database that won't have any impact on anyone else, so you're free to experiment as much as you want.

The example will start with a Windows Forms application. You'll create the model needed to make the database work with SQL Server Express (installed automatically on your system), and then use the resulting model to create a functional application. You'll test the application by managing some data you create with it. The entire process will take an amazingly short time to complete, as described in the following sections.

Starting the Entity Data Model Wizard

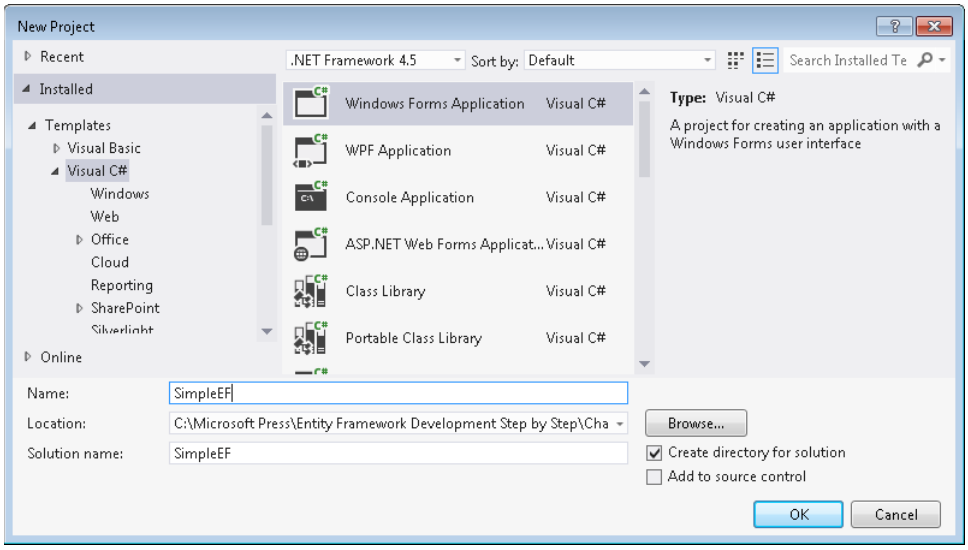
The first step is to create the database model. You can perform this task using a number of methods, most of which developers never use. The easy method is to start the Entity Data Model Wizard and have it do the work for you. That's the approach this example takes, as described in the following steps (you can find this project in the \Microsoft Press\Entity Framework Development Step by Step\Chapter 01\SimpleEF folder of the downloadable source code):

Creating the *SimpleEF* application and adding a database model to it

1. Start Visual Studio 2012.

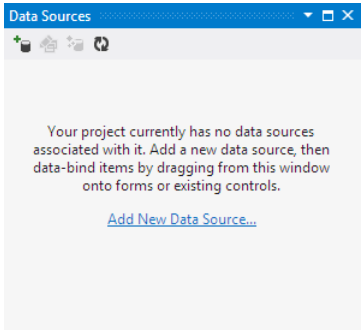
> **Note** This book is designed around Entity Framework 5 and Visual Studio 2012 Professional or above. You could possibly try other versions of Visual Studio, but there is no guarantee that the examples will work. You will most definitely encounter problems trying to work through the examples using any of the Microsoft Express editions of Visual Studio.

2. Choose File | New | Project to display the New Project dialog box, as shown here:

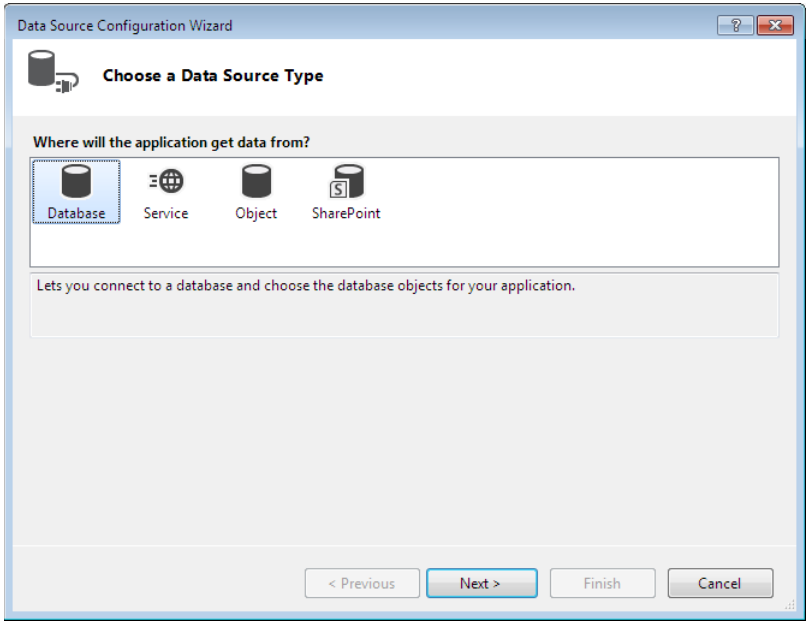


3. Type **SimpleEF** in the Name field and click OK. You'll see a new Windows Forms project.

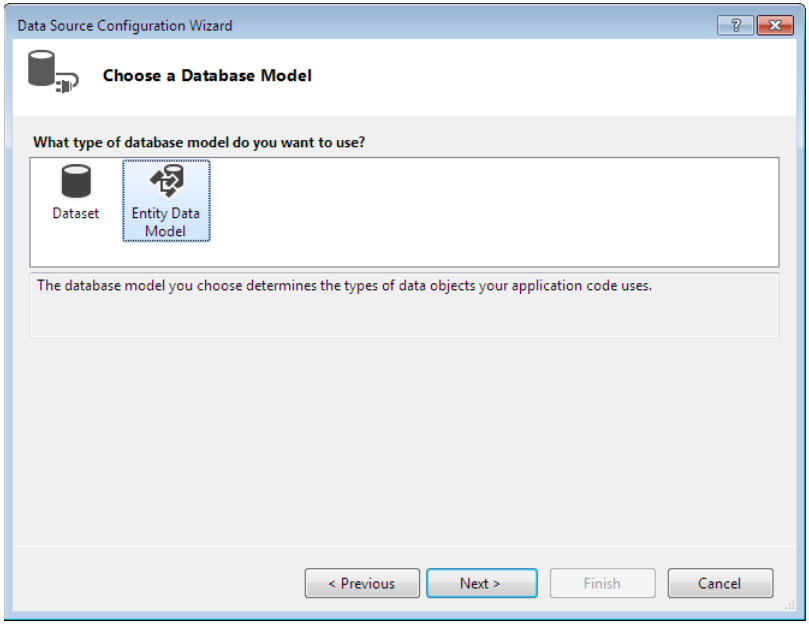
4. Choose View | Other Windows | Data Sources. You'll see the Data Sources window, as shown here:



5. Click Add New Data Source. You'll see the Data Source Configuration Wizard dialog box. The wizard asks you to select a data source type, as shown here:

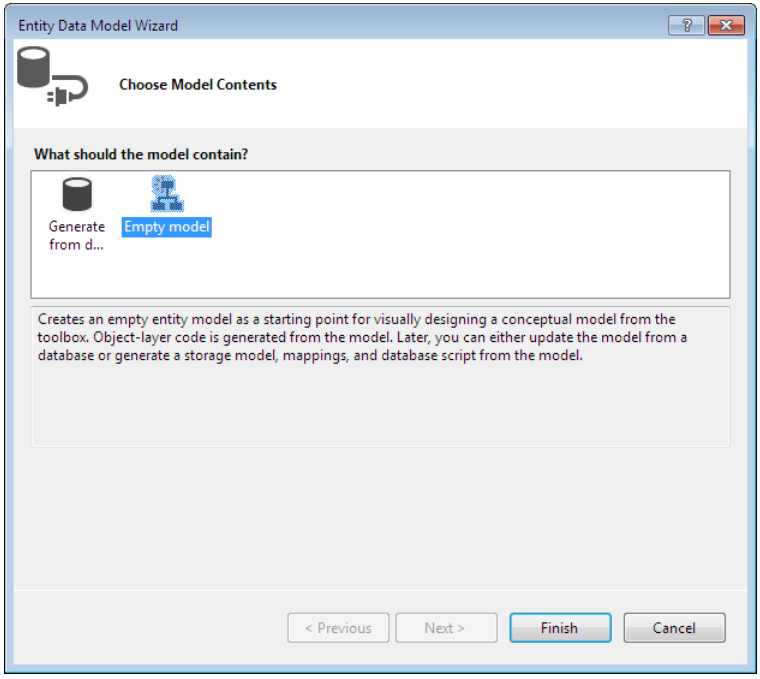


6. Select Database and click Next. The Data Source Configuration Wizard asks you to select a database model, as shown here:

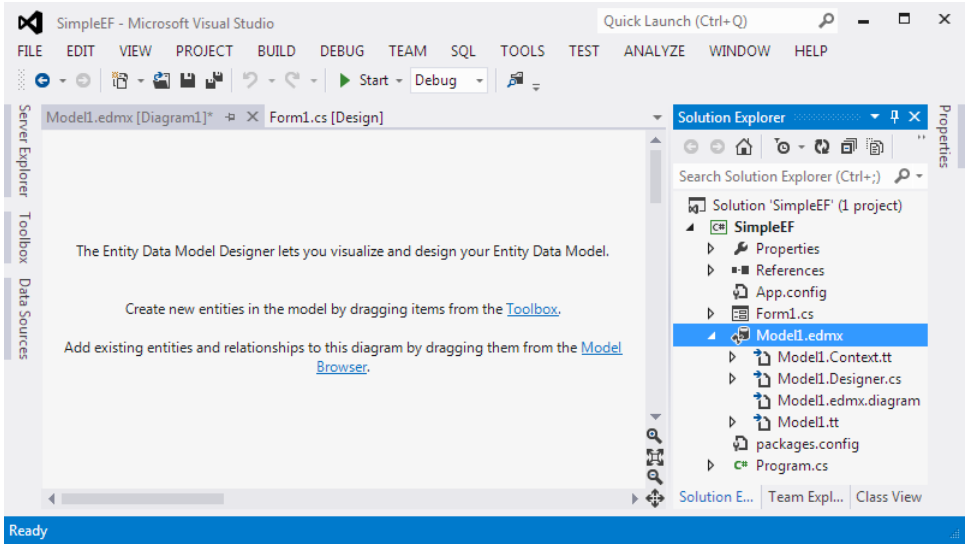


Note The Data Source Configuration Wizard provides access to a number of data source types, not just a database. For example, you could create an application that relies on access to a web service or uses a special kind of object to interact with data. There's also an option to create a data source from your Microsoft SharePoint installation. These other sources are helpful, but discussing them is outside the scope of this book. For the purposes of this book, you work with databases as a data source because the Entity Framework deals with databases, not the other data sources at your disposal.

7. Choose Entity Data Model and click Next. The Data Source Configuration Wizard asks you to choose the model content, as shown here:



8. Choose Empty Model and click Finish. You'll see Visual Studio perform a few tasks. When you have the default User Access Control (UAC) set up, you'll see a Security Warning dialog box telling you that running the script required to generate the Entity Data Model could harm your system. If you see this message, check the Do Not Show This Message Again option and click OK to continue generating the Entity Data Model. It's during this phase of the procedure that you'll see the Entity Data Model Wizard perform the tasks required to generate an empty model for you. After a few additional moments, you'll see a blank Entity Data Model Designer window like the one shown here:

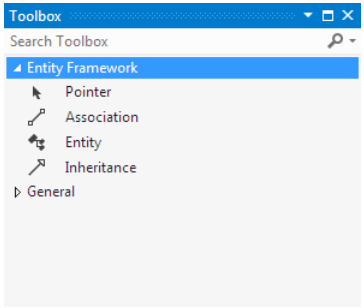


Note When working with existing data, you choose the Generate From Database option instead. The Entity Data Model Wizard will ask you a number of additional questions and create a model based on the existing database, including a full set of diagrams graphically displaying the database structure. Chapter 3 shows how the database-first technique works. For now, just focus on the process used to interact with the Entity Framework.

Solution Explorer also shows the result of adding the new data source. Notice the Model1.EDMX file shown in the screen shot. This file contains the conceptual model, store model, and model mappings. Each feature uses the language (CSDL, SSDL, and MSL) required for that part of the Entity Framework data.

Using the Entity Data Model Designer

After you add an Entity Data Model to your application, you can begin adding items to it from the toolbox—just as you do when adding controls to your application. For example, if you want to add an entity to the model, you drag and drop it onto the Entity Data Model Designer. The toolbox, shown here, contains the elements described earlier in the chapter.

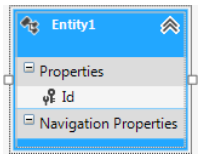


You'll begin working with a model by adding an Entity to it and then configuring the Entity as needed. The example uses a simple Entity named *Customer* with just a few properties that describe the resulting *Customer* object. In this case, you'll use the following properties:

- First Name (*FirstName*)
- Last Name (*LastName*)
- First Address Line (*AddressLine1*)
- Second Address Line (*AddressLine2*)
- City (*City*)
- State/Province (*State_Province*)
- ZIP/Postal Code (*ZIP_Postal_Code*)
- Region/Country (*Region_Country*)

Defining the *SimpleEF* Entity Data Model

1. Drag an *Entity* object from the toolbox to the Entity Data Model Designer. You'll see a new square added containing a blank entity, as shown here:

   Notice that the designer automatically adds an *Id* property for you. This property uniquely identifies a particular entry.

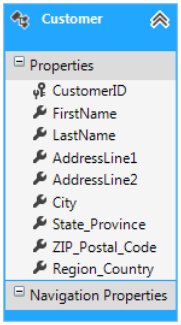
2. Right-click the *Entity1* object and choose Rename from the context menu. The Entity1 entry changes to a text box. Type **Customer** and press Enter.

3. Right-click the *Id* property and choose Rename from the context menu. The *Id* property changes to a text box. Type **CustomerID** and press Enter.

4. Right-click the *Customer* object and choose Add New | Scalar Property from the context menu. You'll see a new property added with the name as a text box.

5. Type **FirstName** (the value shown in parentheses in the previous list) and press Enter.

6. Perform steps 4 and 5 for all of the properties described earlier in this section. When you're finished, your entity should look like this one:

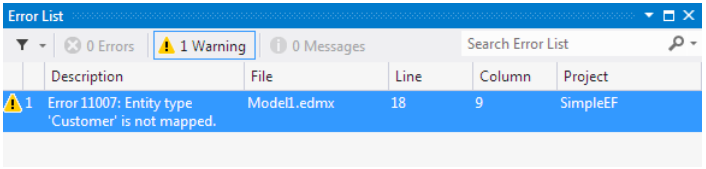


At this point, you could select any of these entity properties and change their properties using the Properties window, just as you would with any application feature. For example, you could change the *Type* property to any of the supported data types. However, for the purposes of this example, you don't actually need to change anything.

Notice that the default *Entity* object color is blue. When working with a complex design, you may want to color code the entities to make them easier to identify. For example, you may want to color customer entities blue and employee entities red. Color coding can make it easier to find the specific entity group you want. To change the color of an entity, select the entity in the designer and change the Fill Color property in the Properties window.

Working with the mapping details

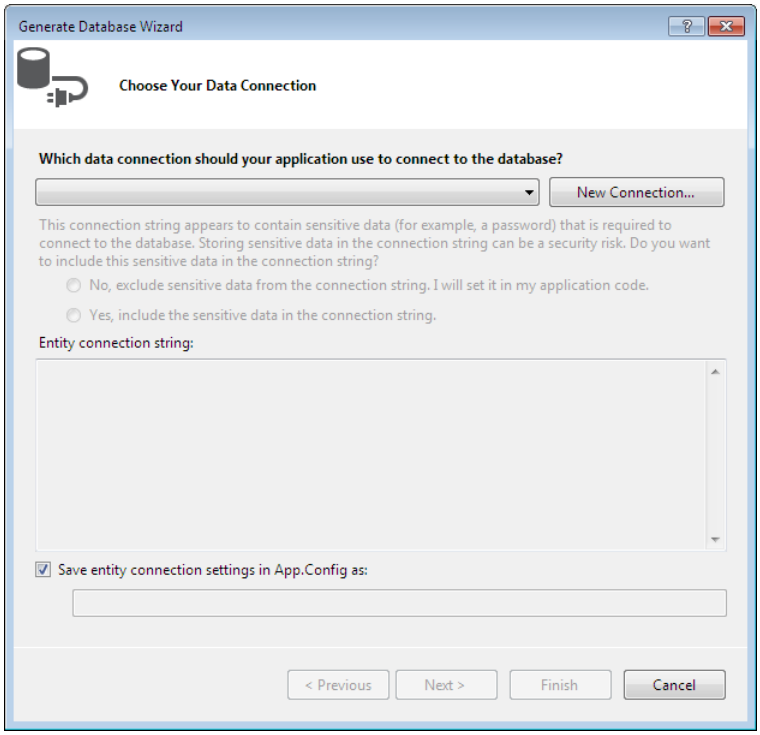
At this point, you've defined a model for the example application. Right-click the *Customer* entity and choose Validate from the context menu. The IDE tells you that entity *Customer* isn't mapped, as shown here:



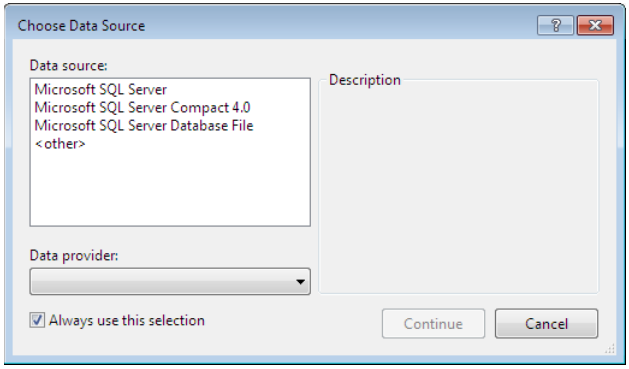
Creating a model doesn't create the required mapping. In fact, the database you just created doesn't exist at all. The model for the database exists, but you still need to tell Visual Studio to interact with the database manager (SQL Server Express in this case) to create the physical database and develop a map between your model and the logical database.

Developing the database and the required mapping

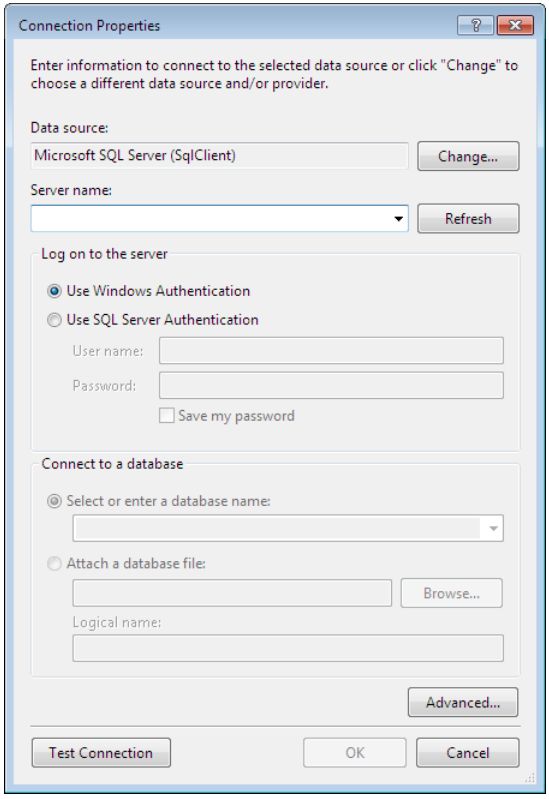
1. Right-click the *Customer* entity and choose Generate Database From Model on the context menu. You'll see the Generate Database Wizard dialog box, as shown here:



2. Click New Connection. You'll see the Choose Data Source dialog box shown here:



3. Select Microsoft SQL Server and then click Continue. You'll see a Connection Properties dialog box like the one shown here:



Note If you plan to work with other database managers, make sure you clear the Always Use This Selection check box. Doing so ensures that Visual Studio displays this dialog box each time so that you can choose which database manager you want to use.

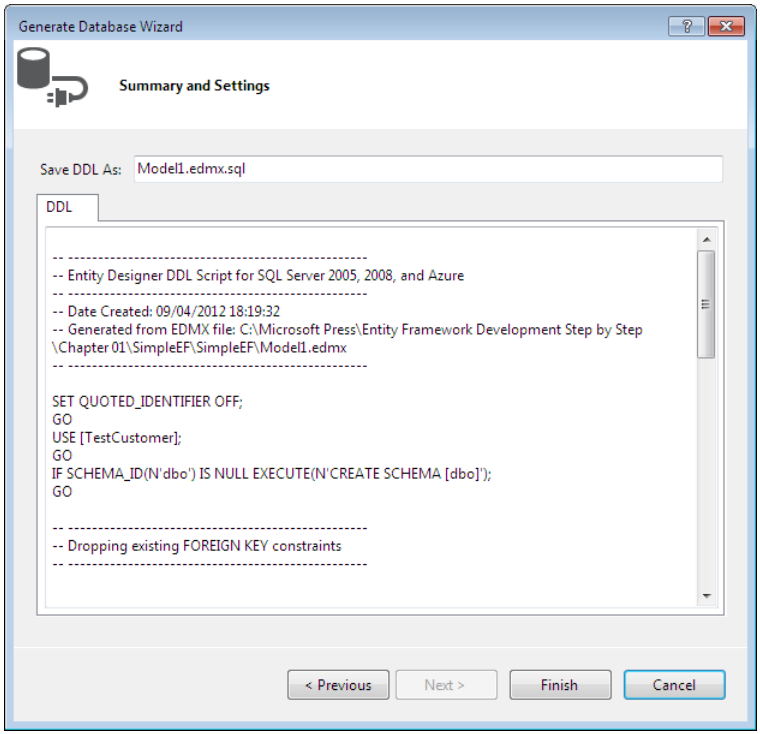
4. Choose the name of the server you want to use in the Server Name drop-down list box.
5. Type **TestCustomer** in the Select Or Enter A Database Name field.

Note If you click Test Connection at this point, you should see an error message stating the database doesn't exist. That's because Visual Studio hasn't created it yet. The database will exist after these steps are complete.

6. Click OK. You'll see a dialog box telling you that the database doesn't exist. Visual Studio asks permission to attempt to create the database for you.

7. Click Yes. Visual Studio creates the new database for you. This is a blank database—it doesn't contain any tables, views, indexes, or anything else normally associated with a database. You'll return to the Generate Database Wizard dialog box. However, now the connection information is filled in.

8. Click Next. The Generate Database Wizard creates the Data Definition Language (DDL) script required to create everything in the model you designed, as shown here. You can scroll through this script to see the SQL statements used to make your model a real database and associated table.



9. Click Finish. You'll see the script, Model1.EDMX.sql, open. It hasn't executed yet. All that the Generate Database Wizard has done is create the script required to make your database model functional.

10. Choose SQL | Transact-SQL Editor | Execute. You'll see a connection dialog box where you can enter the information required to connect to the SQL Server instance you've selected.

11. Enter any required credentials and click Connect. Visual Studio connects to the database manager and executes the SQL script it created. At this point, your database is ready for use. Notice that you didn't have to access the database manager yourself or create any scripts by hand.

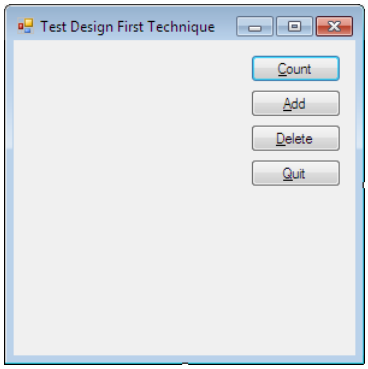
Using the resulting framework to display data

Now that you have a database to use—a database generated from a model you created—you might want to see the database in action. There are a number of ways to accomplish the task, but for this first sample, it's probably best to try something easy. The one piece of information you absolutely need to know before you start is that the model you created earlier also generated code. Part of this code is the creation of a container that you use to access the database. The container class always starts with the name of the model, followed by the word *container*. For this example, this means that the name of the container class is *Model1Container*.

Nothing else you do with the Entity Framework is going to be outside your experience if you've worked with collections in the past. The following steps create a simple application that will test just a few of the features that this model provides. Chapter 2, "Looking more closely at queries," will help you start performing more complex tasks.

Creating an application based on the *TestCustomer* database

1. Add four buttons to the Windows Forms application you created at the outset of this example, and name them *btnCount*, *btnAdd*, *btnDelete*, and *btnQuit*. Here's an example of the simple form as it appears in the downloadable source:



2. Right-click the Form1.cs entry in Solution Explorer and choose View Code from the context menu. You'll see the Code Editor. Add a reference to the model container and instantiate it in the form's constructor, as shown here:

```
// Define a container to hold the database information.
Model1Container ThisContainer;

public Form1()
{
   InitializeComponent();

   // Instantiate the container.
   ThisContainer = new Model1Container();
}
```

ThisContainer contains a reference to all of the elements found in the model. In this case, the model only contains a reference to one table, *Customers*. However, in a production application, you could use *ThisContainer* to access every table, view, index, or other feature in the database.

3. Double-click Count. Visual Studio creates an event handler for you. Add the following code to the event handler:

```
private void btnCount_Click(object sender, EventArgs e)
{
   // Display the number of database records.
   MessageBox.Show("There are " +
      ThisContainer.Customers.Count().ToString() +
      " Records.");
}
```

The container for all of the database elements is found in *ThisContainer*. Within the container is a table named *Customers*. The *Count()* method outputs the number of records in the specified table.

4. Double-click Add and add the following code to the resulting event handler:

```
private void btnAdd_Click(object sender, EventArgs e)
{
   // Create a new record.
   Customer ThisCustomer = ThisContainer.Customers.Create();

   // Add some random data.
   Random ThisValue = new Random(DateTime.Now.Millisecond);
   ThisCustomer.FirstName = ThisValue.Next().ToString();
   ThisCustomer.LastName = ThisValue.Next().ToString();
   ThisCustomer.AddressLine1 = ThisValue.Next().ToString();
   ThisCustomer.AddressLine2 = ThisValue.Next().ToString();
   ThisCustomer.City = ThisValue.Next().ToString();
   ThisCustomer.State_Province = ThisValue.Next().ToString();
   ThisCustomer.ZIP_Postal_Code = ThisValue.Next().ToString();
   ThisCustomer.Region_Country = ThisValue.Next().ToString();

   // Add a new record.
   ThisContainer.Customers.Add(ThisCustomer);
   ThisContainer.SaveChanges();

   // Inform the user.
   MessageBox.Show("Added " + ThisCustomer.CustomerID.ToString());
}
```

The example begins by creating a new *Customer* record, *ThisCustomer*. It then fills the fields with random numeric values. The content is simply there to make it easy to view the record information later.

In order to add the new record to the database, the example calls the *ThisContainer.Customers.Add()* method. This method requires a *Customer* object as input. The changes won't take effect until the application calls *ThisContainer.SaveChanges()*. You need to make sure your

code calls the *SaveChanges()* method regularly; otherwise, you risk losing application data. Finally, the application shows the record number added to the application.

5. Double-click Delete and add the following code to the resulting event handler:

```
private void btnDelete_Click(object sender, EventArgs e)
{
   // Obtain the first record.
   Customer ThisCustomer = null;
   if (ThisContainer.Customers.Count() > 0)
      ThisCustomer = ThisContainer.Customers.First();
   else
   {
      // Display an error message if there are no records to delete.
      MessageBox.Show("No Records to Delete");
      return;
   }

   // Delete it.
   ThisContainer.Customers.Remove(ThisCustomer);
   ThisContainer.SaveChanges();

   // Inform the user.
   MessageBox.Show("Deleted " + ThisCustomer.CustomerID.ToString());
}
```

A production application would have a lot more checks than this one does, but the code begins by checking whether there are any records to delete in the *Customers* table. If not, the event handler exits after providing an error message.

There are a number of ways to obtain a record from the *Customers* table. For that matter, you might simply want to search for a particular record based on some criterion and delete all those that match. In this case, the code uses the *ThisContainer.Customers.First()* method to obtain a copy of the first record in the table. The code then calls *ThisContainer.Customers.Remove()* to remove the record and *ThisContainer.SaveChanges()* to make the changes permanent. The code then informs the user about the deletion and displays the ID of the customer it deleted.

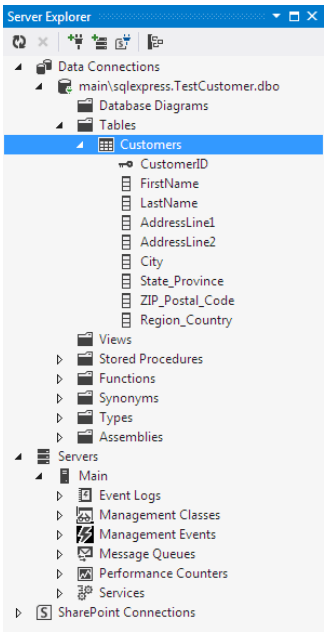
6. Double-click Quit and add the following code to the resulting event handler:

```
private void btnQuit_Click(object sender, EventArgs e)
{
   // Save the database.
   ThisContainer.SaveChanges();

   // End the program.
   Close();
}
```

One task you should always perform before you exit the application is to save the database changes one more time—just to ensure that none of the changes are lost. After the code calls *ThisContainer.SaveChanges()*, it exits by closing the form.

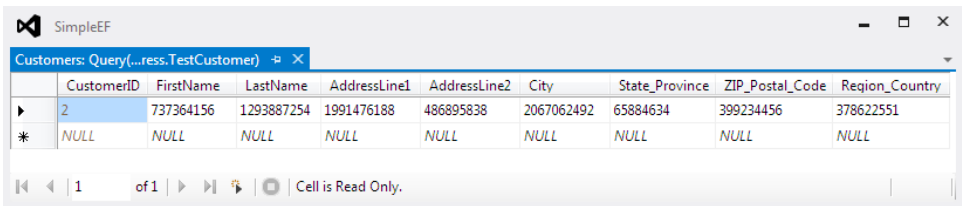
7. Click Start and try some of the buttons. For example, click Count and you'll see the current record count (0 if there are no records). Click Add and you'll see the identifier of the customer that the application has added. Likewise, click Delete and you'll see the identifier of the customer that the application has deleted. Make sure you end up with at least one record in the database.

8. Choose View | Server Explorer. You'll see the Server Explorer window shown here:



9. Drill down into the TestCustomer.dbo\Tables\Customers entry, as shown in the preceding image. Notice that the complete table structure is precisely as you designed it.

10. Right-click Customers and choose Show Table Data from the context menu. You'll see a new window appear with the data from the table as shown here (your data will most definitely differ from mine because the data is randomly generated in this application):



This environment is fully interactive, so you can use it to check the results of your database experiments. More importantly, you can use it to modify the data as necessary to meet test requirements.

11. Click Quit to end the application. You can always experiment with this application later.

Getting started with the Entity Framework

The Entity Framework makes it possible to write database applications using less manually written code because the Entity Framework relies on the content of the conceptual model, storage model, and mapping model files to automatically generate classes that an application can use to access the database reliably. The use of the Entity Framework makes developers more productive and generally reduces application errors. In addition, the automation that the Entity Framework provides helps ensure that the application remains up to date. Changes made by the developer or DBA are automatically reflected in the application.

This chapter has introduced you to the Entity Framework. Make sure you understand the three layers—conceptual model, storage model, and model mapping—before you proceed to Chapter 2. Also take time to create the sample application and view the files it creates. The more time you spend interacting with the data that the Entity Framework creates and manages, the better. Of course, all of the work of creating classes is done for you in the background, but it's still a good idea to know the source of the automation and have an idea of how it works for those situations where the automation doesn't quite produce the results you expected. As part of working with this chapter, try creating your own project based on data that you already have in a sample database on your system. (Please don't work with any production data until you're proficient with the Entity Framework.)

Chapter 2 takes the next natural step in working with the Entity Framework. Instead of simply creating a project and viewing the resulting files, you're going to begin working with some data by making queries. After all, data stored in a database isn't useful until you can get it out and display it to an end user in a useful form. Once you complete Chapter 2, you may want to come back to this chapter and use the techniques described here to view the files that the sample in that chapter creates. You'll see some differences because now you'll be interacting with the data in a meaningful way. Viewing the differences will add to your knowledge of how the Entity Framework interacts with the database and generates XML to model it.

Chapter 1 quick reference

To	Do this
See how the application views the database	Open the .CSDL or .EDMX file and view its content.
See how the database manager views the database	Open the .SSDL or .EDMX file and view its content.
Determine how the Entity Framework resolves differences between the application view and the database manager view of the database	Open the .MSL or .EDMX file and view its content.
Create a new conceptual model	Click Add New Data Source in the Data Source window and choose Empty Model when prompted.
Add entities to the new conceptual model	Drag and drop an *Entity* object from the Entry Framework folder of the toolbox to the Entity Data Model Designer.
Generate a physical database based on your design	Right-click the entity you want to work with and choose Generate Database From Model on the context menu.
Generate the tables and other elements in your model	Choose SQL | Transact-SQL Editor | Execute.
Use the new database in an application	Create a reference to the model container, such as *Model1Container ThisContainer = new Model1Container();*, where *Model1* is the name of the model you want to use.

CHAPTER 6

Manipulating data using LINQ

After completing the chapter, you'll be able to

- Describe the basics of LINQ to Entities functionality.
- Specify how LINQ statements are compiled.
- List and use the essential LINQ to Entities functions.

In most cases, developers with a strong C# background, but without an equally strong database background, use Language Integrated Query (LINQ) to query the databases they create and manage using the Entity Framework. LINQ to Entities offers a number of benefits to developers, but the main benefit is simplicity. It's possible to create relatively complex queries without knowing much about the underlying database from a DBMS perspective. Developers can also use syntax that's familiar to make the query, rather than resorting to working with SQL. In addition, the compiler performs part of the work of interacting with the database for the developer, so that the developer can focus on the data-set and not on the language used to access it. In short, the developer gains a considerable efficiency advantage using LINQ to Entities.

This chapter begins by introducing you to LINQ to Entities. You need to know something about how LINQ to Entities works, and you also need to know the syntax so that you can make queries. The chapter won't provide an extensive reference, but you'll have enough information to perform common tasks and a few advanced tasks. The point is that you'll have the information required to get started using LINQ to Entities to perform useful work. The material provided will help you understand the examples better.

Tip There are actually two syntaxes you can use to formulate a LINQ query: query and method based. The *query* expression syntax tends to be easier to understand and clearer, so that's the form used in this book whenever possible. The *method-based* expression syntax is more flexible, and you can perform a few tasks using it that you can't with the query expression syntax, so the book will use this form when necessary to perform complex tasks. Presenting the examples this way will help you better understand when you need to use one syntax over the other. You can also read a comparison of the two syntaxes at *http://msdn.microsoft.com/library/bb397947.aspx*.

As with any LINQ query, LINQ to Entities queries are compiled to determine what they actually mean. The compiler takes the query you create and turns it into something that .NET understands. The next section of this chapter discusses how this process occurs and how it affects the way you use LINQ to Entities. This part of the chapter also provides a few insights into when you need to use the method-based expression syntax to obtain the output you desire.

The final part of the chapter discusses how to use LINQ to Entities with both entity and database functions, which, after all, is the entire point of working with LINQ to Entities in the first place. This section provides you with examples you can use to better understand how LINQ to Entities works. In addition, this material sets the stage for future examples in the book. When you finish this section, you'll have the knowledge needed to move on to the more advanced examples in the book.

Note LINQ to Entities is just one form of a more complex product that appears under the LINQ umbrella. There are, in fact, many different forms of LINQ you can use. However, once you know how to use one form of LINQ, you essentially know how to use them all. That's one of the beauties of a declarative language—you focus on what you need, rather than how to obtain it. You can find a general overview of LINQ as a product at *http://msdn.microsoft.com/library/bb308959.aspx*.

Introducing LINQ to Entities

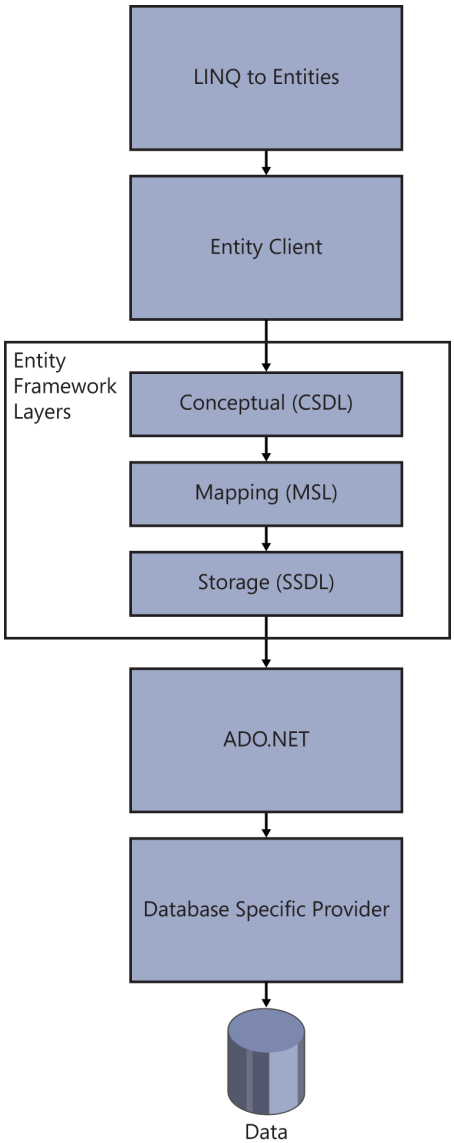
One of the most important concepts to understand about LINQ to Entities is that it's a declarative language. The focus is on defining what information you need, rather than on how to obtain the information. This means that you can spend more time working with data and less time trying to figure out the underlying code required to perform tasks such as accessing the database. It's important to understand that declarative languages don't actually remove any control from the developer; rather, they help the developer focus attention on what's important.

The sections that follow provide you with a basic overview of LINQ to Entities. You learn about how the LINQ to Entities provider, *EntityClient*, works, discover how to create a basic query, and then move on to some reference information you need later to work with LINQ to Entities in examples. These sections will continue to be useful as a reference as you progress through the book, so keep them in mind as you move on to other topics later.

Considering the LINQ to Entities provider

When working with LINQ to Entities, you rely on a new provider named *EntityClient*. LINQ to Entities transforms your query into *EntityClient* objects and method calls. The *EntityClient* provider then creates an interface between the LINQ to Entities queries and the underlying Microsoft ADO.NET providers through the various layers of the Entity Framework. The *EntityClient* interacts directly with the conceptual model, as shown in the following graphic.

Warning A number of drawings and discussions available online don't mention the need for a database-specific provider. If you're using a DBMS other than Microsoft SQL Server or one of the compatible DBMSs described in Chapter 1, "Getting to know the Entity Framework," then you'll find that your queries won't work. You still depend on ADO.NET to complete tasks.



You don't create an *EntityClient* directly. Instead, you indirectly work with the members of the *System.Data.EntityClient* namespace (see *http://msdn.microsoft.com/library/system.data.entityclient.aspx* for details). In order to start a session with a database, the application creates a connection to it with the *EntityConnection* object. It then transmits queries and other requests using an *EntityCommand* and reads the results using an *EntityDataReader*. When you work with LINQ to Entities, the compiler generates the necessary code for you—the focus for you as a developer is the query declaration, rather than the actual code used to make the calls. However, it's important to know what happens in the background.

The standard ADO.NET providers are still used to communicate with the database. However, you don't need to worry about writing all of the code used to perform this communication; *EntityClient* performs this task for you. A simple way to look at *EntityClient* is as a translator that takes your declarative language query and puts it into terms that ADO.NET can understand.

The LINQ to Entities provider interacts with ADO.NET directly, which means that you don't need any other provider to use LINQ to Entities with other databases. However, ADO.NET uses database-specific providers. Microsoft Visual Studio ships with ADO.NET providers for both SQL Server and SQL Server Compact. Of course, there are other databases on the market. You can find a number of ADO.NET providers for other databases at *http://msdn.microsoft.com/data/dd363565.aspx*. If you don't find a suitable provider on MSDN, try other sites, such as Devart (*http://www.devart.com/linqconnect/*) and SQLite (*http://www.sqlite.org/*).

Developing LINQ to Entities queries

There are a number of ways to formulate LINQ queries. The use of different approaches provides developers with flexibility and enables a developer to code using the style that the developer is used to. The first division in LINQ queries is the syntax. A developer has a choice between using the query expression syntax or the method-based expression syntax. Of the two, the query expression syntax is the easiest to understand, while the method-based expression syntax offers the greatest flexibility.

It's also possible to specify precise output type or to allow the compiler to derive the output type based on the query you create (an implicit type). A *precise* output type means providing a specific type, such as *IQueryable<Customers>*. A *derived* output type relies on the *var* keyword (see *http://msdn.microsoft.com/library/bb383973.aspx* for a detailed description of this keyword). The compiler determines the variable type for you. The precise output type provides you with additional control over how the query is made and the results it provides. Using the *var* keyword is necessary at times because you may not be able to determine the precise type. In addition, the *var* keyword makes it more likely that the query will succeed and provide usable data, because the compiler determines the correct type for you.

The query itself requires the use of keywords or methods that reflect those keywords. When using the query expression syntax, a query will use the *select*, *in*, and *from* keywords as a minimum. The best way to see how this works is through an example. The following procedure relies on the *ModelFirst* example you created in the "Creating a model-first example" section of Chapter 3, "Choosing a workflow." (You can find this example in the \Microsoft Press\Entity Framework Development Step by Step\Chapter 06\ModelFirst (LINQ Query) folder of the downloadable source code.)

Creating a LINQ to Entities query

1. Copy the *ModelFirst* example you created in Chapter 3 to a new folder and use this new copy for this example (rather than the copy you created in Chapter 3).
2. Add a new button to *Form1*. Name the button *btnQuery* and set its *Text* property to *&Query*.
3. Double-click *btnQuery* to create a new click event handler.
4. Type the following code for the *btnQuery_Click()* event handler:

```
private void btnQuery_Click(object sender, EventArgs e)
{
    // Create the context.
    Rewards2ModelContainer context = new Rewards2ModelContainer();

    // Obtain the customer list.
    var CustomerList =
        from cust in context.Customers
        select cust;

    // Process each customer in the list.
    StringBuilder Output =
        new StringBuilder("Customer List:");
    foreach (var Customer in CustomerList)
    {
        // Create a customer entry for each customer.
        Output.Append("\r\n" + Customer.CustomerName +
            " has made purchases on: ");

        // Process each purchase for that particular customer.
        foreach (var Purchase in Customer.Purchases)
            Output.Append("\r\n\t" + Purchase.PurchaseDate);
    }

    // Display the result on screen.
    MessageBox.Show(Output.ToString());
}
```

The example begins by creating a context. It's important to remember that you still need to create this connection to the Entity Framework layers in order to access the database. The LINQ query will be translated by the *EntityClient* into a series of commands that will interact with the context to perform the tasks you specify.

The LINQ query comes next. Notice that the example is using the *var* keyword rather than a specific type. The example asks for the list of customers from the context and places each customer in *cust*. It then selects *cust* and places this value in *CustomerList*. Hover the mouse over *CustomerList* in the *foreach* loop that follows, and you'll see that Visual Studio really does assign it a type of *IQueryable<Customers>*, as shown here:

```
// Process each customer in the list.
StringBuilder Output =
    new StringBuilder("Customer List:");
foreach (var Customer in CustomerList)
{
    // Create a customer
    Output.Append("\r\n" + Customer.CustomerName +
        " has made purchases on: ");
```

(local variable) IQueryable<Customers> CustomerList

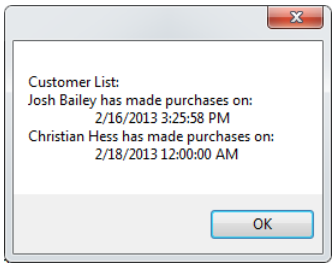
Let's say that you decide you want to use *IEnumerable* instead of *IQueryable* (see the "Determining when to use *IEnumerable* in place of *IQueryable*" sidebar for details). In order to use *IEnumerable*, you'd need to rewrite the query like this:

```
// Obtain the customer list.
IEnumerable<Customers> CustomerList =
    from cust in context.Customers
    select cust;
```

This is a master/detail database setup, so the example creates two *foreach* loops to process the data. The first *foreach* loop obtains one Customer from *CustomerList* and processes the customers one at a time. The second *foreach* loop obtains one Purchase from *Customer.Purchases* and processes each purchase for that customer one at a time. The result is an output string that is displayed in a message box.

5. Click Start or press F5. The application compiles and runs.

6. Click Query. You'll see the result shown here (assuming that you ran the example from Chapter 3 and didn't modify the code from that example):



Note Most of the information you see in the dialog boxes in this chapter will match those on your system. However, you'll encounter a few differences, such as dates. In addition, the application dialog boxes may not match precisely. These small differences won't make any difference in the performance of the example applications.

Determining when to use *IEnumerable* in place of *IQueryable*

When working with LINQ to Entities, some developers assume that you should always use *IQueryable* because it derives from *IEnumerable* and therefore must be superior. Actually, the two interfaces have specific purposes and you should employ the one that works best for your particular need. There are quite a few differences between the two, but here are some general rules of thumb you can follow:

- ***IEnumerable*** Provides a forward-only in-memory presentation of data. Because the query is executed immediately and completely, your application will see a performance boost during the enumeration process when the user is most apt to see the difference. Working with *IEnumerable* means that your application uses *Func* objects that result in the query being executed immediately. You can read more about *Func* objects at *http://msdn.microsoft.com/library/bb534960.aspx*.
- ***IQueryable*** Provides remote access to a database or a web service and allows both forward and reverse iteration. Use this form to enhance the flexibility of your application and its ability to work with remote sources, especially web services. Working with *IQueryable* means that your application uses *Expression* objects that result in the query being executed only when the application requests an enumeration. Because the query is delayed, an *IQueryable* object can perform certain optimizations when using a where or other clause that would throw out some of the results that would normally be processed by an *IEnumerable* object. The tradeoff is that you save memory and some network bandwidth in exchange for longer enumeration times. You can read more about *Expression* objects at *http://msdn.microsoft.com/library/system.linq.expressions.aspx*.

Using the correct object type for the situation can improve the efficiency of your application. It's important to consider how your application works when making the choice. When in doubt, *IQueryable* is the preferred choice because it does offer greater flexibility, and the performance benefits of *IEnumerable* could be outweighed by the amount of data retrieved over high-cost network *connections*. When creating a query that includes a *where* clause, the costs of using *IEnumerable* quickly make *IQueryable* the better choice. *IEnumerable* is almost always a better choice when making a straightforward query, like the one in the example, because the example uses all of the results anyway.

Defining the LINQ to Entities essential keywords

It's important to know the basic keywords used to create a LINQ query. Interestingly enough, there are only a few keywords to remember, but you can combine them in various ways to obtain specific results. The following list contains these basic keywords and provides a simple description of each one (future examples will expand on these definitions for you):

- ***ascending*** Specifies that a sorting operation takes place from the least (or lowest) element of a range to the highest element of a range. This is normally the default setting. For example, when performing an alphabetic sort, the sort would be in the range from *A* to *Z*.
- ***by*** Specifies the field or expression used to implement a grouping. The field or expression defines a key used to perform the grouping task.
- ***descending*** Specifies that a sorting operation takes place from the greatest (or highest) element of a range to the lowest element of a range. For example, when performing an alphabetic sort, the sort would be in the range from *Z* to *A*.
- ***equals*** Used between the left and right clauses of a *join* statement to join the primary contextual data source to the secondary contextual data source. The field or expression on the left of the *equals* keyword specifies the primary data source, while the field or expression on the right of the *equals* keyword specifies the secondary data source.
- ***from* (required)** Specifies the data source used to obtain the required information and defines a range variable. This variable has the same purpose as a variable used for iteration in a loop.
- ***group*** Organizes the output into groups using the key value you specify. Use multiple group clauses to create multiple levels of output organization. The order of the *group* clauses determines the depth at which a particular key value appears in the grouping order. You combine this keyword with *by* to create a specific context.
- ***in* (required)** Used in a number of ways. In this case, the keyword determines the contextual database source used for a query. When working with a join, the in keyword is used for each contextual database source used for the *join*.
- ***into*** Specifies an identifier that you can use as a reference for LINQ query clauses such as *join*, *group*, and *select*.

Warning A common error that some developers make is to confuse the *into* keyword with the *in* keyword. The *into* keyword serves an entirely different purpose, and using it in place of the *in* keyword will cause an error.

- ***join*** Creates a single data source from two related data sources, such as in a master/detail setup. A join can specify an inner, group, or left-outer join, with the inner join as the default. You can read more about joins at *http://msdn.microsoft.com/library/bb311040.aspx*.
- ***let*** Defines a range variable that you can use to store subexpression results in a query expression. Typically, the range variable is used to provide an additional enumerated output or to increase the efficiency of a query (so that a particular task, such as finding the lowercase value of a string, need not be done more than one time).

- ***on*** Specifies the field or expression used to implement a join. The field or expression defines an element that is common to both contextual data sources.
- ***orderby*** Creates a sort order for the query. You can add the *ascending* or *descending* keyword to control the order of the sort. Use multiple *orderby* clauses to create multiple levels of sorting. The order of the *orderby* clauses determines the order in which the sort expressions are handled, so using a different order will result in different output.
- ***where*** Defines what LINQ should retrieve from the data source. You use one or more Boolean expressions to define the specifics of what to retrieve. The Boolean expressions are separated from each other using the *&&* (AND) and || (OR) operators.
- ***select* (required)** Determines the output from the LINQ query by specifying what information to return. This statement defines the data type of the elements that LINQ returns during the iteration process.

Defining the LINQ to Entities operators

The keywords described in the "Defining the LINQ to Entities essential keywords" section of the chapter determine what happens when a query is made using the query expression syntax. Operators determine how the query is made when using the method-based expression syntax. You use operators to modify the output in the following ways:

- **Sort** Modify the natural order of the data returned from the data source. For example, you could create a sorted order of customers based on their last name, even if the database keeps the customer list in a random order.
- **Group** Create an order that is depending on a specific field or expression. For example, you could group a list of customers by the first letter of their last name.
- **Shape** Modify the natural appearance of the data to obtain specific results. For example, you could filter the data so that the output only contains customers whose last name begins with a *G*, or you could determine the average value of the data using aggregation.

The following sections describe a number of common tasks you can perform using LINQ to Entities operators. These are basic operations. Remember that you can combine operators to create almost any data manipulation scenario. Using LINQ to Entities operators makes it possible for you to declare what you want as output, rather than determine how to obtain it. The compiler determines how a particular task is done.

Note LINQ to Entities supports most, but not all, of the standard LINQ methods. For example, you can use a *Select* method with this signature:

```
IQueryable<TResult> Select<TSource, TResult>(
this IQueryable<TSource> source,
Expression<Func<TSource, TResult>> selector
)
```

But you can't use a *Select* method with this signature:

```
IQueryable<TResult> Select<TSource, TResult>(
this IQueryable<TSource> source,
Expression<Func<TSource, int, TResult>> selector
)
```

The difference is subtle. Notice that the second signature includes an *int* as part of the *Func* declaration, which means you can't use the index of the element, as described at *http://msdn.microsoft.com/library/system.linq.enumerable.select.aspx*. You can see a complete list of the supported and unsupported methods at *http://msdn.microsoft.com/library/bb738550.aspx*.

Performing filtering and projection

The main task of any LINQ to Entities expression is to obtain data and provide it as output. The "Developing LINQ to Entities queries" section of this chapter demonstrates the techniques for performing this basic task. However, once you have the data, you may want to project or filter it as needed to shape the data prior to output.

Projection is the act of modifying the output to shape it in a specific way. For example, you can change the case of the characters in a string or perform a calculation on numeric output. It's also possible to use methods to transform the data in a variety of ways that are only limited by your imagination and the requirements of your application. The methods associated with projection are *Select()* and *SelectMany()*.

Filtering is the act of removing undesirable elements from the output. You may only want the names of customers who have achieved a certain number of sales or who live in a particular area. Use the *Where()* method to achieve the desired level of filtering.

Note LINQ to Entities supports all of the common LINQ methods associated with filtering and projection, except for those that require a positional (indexing) argument.

Performing joins

Look again at the example in the "Developing LINQ to Entities queries" section of this chapter. Notice that the example is able to obtain the list of purchases associated with a particular customer because there is a navigable property that is defined as part of the model. It's important to keep this bit of

information in mind, especially when you normally work with SQL Server directly by making SQL statements. The join defined by LINQ to Entities is for related tables that have no navigable properties in the model. The result is the same as a standard join, but the purpose of the join is different. Use navigable properties whenever possible to work with related tables.

When performing a join to group like tables together, you use the *Join()* or *GroupJoin()* method. The tables must still possess a common attribute or property that you can exploit to create the relationship. For example, let's say that your in-house database has a table containing a list of products that employ a bar code for identification. However, the description of the product resides on a web service hosted by the supplier. You can use a join on the bar code to obtain a description for the product in your in-house database from the supplier's web service. Because you don't support or own the supplier's database, the database won't appear as part of your model, and you won't have any navigable properties to access it.

Note The LINQ to Entities *Join()* and *GroupJoin()* methods provide full support for all of the standard LINQ overrides, except those that require use of the *IEqualityComparer* interface. This is because LINQ to Entities can't translate the comparer to the source database. You can read more about *IEqualityComparer* at *http://msdn.microsoft.com/library/ms132151.aspx*.

Creating a set

Shaping a result set means defining the set according to specific properties. For example, you might only want the distinct elements from the result set of a query. Even though a particular row in a table is distinct, the result set may not contain the entire row, resulting in duplicates in the output, so you need a way to shape the output so the user only sees unique entries. The methods for creating sets are *All()*, *Any()*, *Concat()*, *Contains()*, *DefaultIfEmpty()*, *Distinct()*, *EqualAll()*, *Except()*, *Intersect()*, and *Union()*.

Note The LINQ to Entities set-related methods provide full support for all of the standard LINQ overrides, except those that require use of the *IEqualityComparer* interface. This is because LINQ to Entities can't translate the comparer to the source database. You can read more about *IEqualityComparer* at *http://msdn.microsoft.com/library/ms132151.aspx*.

Ordering the output

Sorting a result set modifies the order in which the individual records appear so that the user can more easily detect patterns in the output, find a specific output, and look for errors, such as misspellings and duplicate entries. You can combine ordering methods to create a unique output. However, it's an error to provide the same ordering methods more than one time on a result set, and you'll see an exception if you try to do so. The ordering methods are *OrderBy()*, *OrderByDescending()*, *ThenBy()*, *ThenByDescending()*, and *Reverse()*.

When ordering a result set, it's important to realize that LINQ to Entities works against the data source, rather than using an in-memory representation, as would be done when working with the Common Language Runtime (CLR) objects. The data source may have special sort functionality implemented, such as case ordering, kanji ordering, and null ordering. The difference in sort functionality will affect the output you see.

Note The LINQ to Entities ordering-related methods provide full support for all of the standard LINQ overrides, except those that require use of the *IComparer* interface. This is because LINQ to Entities can't translate the comparer to the source database. You can read more about *IComparer* at *http://msdn.microsoft.com/library/8ehhxeaf.aspx*.

Grouping the output

Sorting a result by grouping like items together using a common attribute (such as all customers who live in a particular city) helps users see patterns in the output. When grouping like items together, you use the *GroupBy()* method. It's possible to create multiple levels of grouping by combining multiple *GroupBy()* method calls. Unlike sorting methods, you can create multiple levels of the same *GroupBy()* method calls because each *GroupBy()* method call creates a new level in the output.

When grouping a result set, it's important to realize that LINQ to Entities works against the data source, rather than using an in-memory representation, as would be done when working with the CLR objects. The data source may contain null values that will affect the output in ways that you don't see when performing the same task using CLR objects.

Note The LINQ to Entities *GroupBy()* method provides full support for all of the standard LINQ overrides, except those that require use of the *IEqualityComparer* interface. This is because LINQ to Entities can't translate the comparer to the source database. You can read more about *IEqualityComparer* at *http://msdn.microsoft.com/library/ms132151.aspx*.

Performing aggregation

Shaping the result set by combining or aggregating it in certain ways can help a user see the information in a new way. For example, you might obtain the average of a numeric field so that the user knows when a particular entry is either higher or lower than average. The methods you use to aggregate data are *Aggregate()*, *Average()*, *Count()*, *LongCount()*, *Max()*, *Min()*, and *Sum()*.

There are some significant differences in the way that aggregation occurs when using LINQ to Entities, as contrasted to using the CLR. The most important difference is that the calculations occur on the server, so any loss of precision or type conversions will occur on the server as well. When an error occurs, such as an overflow, the exception is raised as a data source or Entity Framework exception, rather than a standard CLR exception. The errors are only raised when they conflict with the data source assumptions about the data. For example, when working with null values, a CLR calculation

will raise an error, but SQL Server won't. Table 6-1 describes how SQL Server handles nulls so that you know what to expect as output.

TABLE 6-1 Techniques SQL Server uses to handle nulls

Method	No data	All nulls	Some nulls	No nulls
Average	Returns null	Returns null	Returns the average of the non-null values in the sequence	Returns the average of all of the values in the sequence
Count	Returns 0	Returns the number of null values in the sequence	Returns the combined number of null and non-null values in the sequence	Returns the total number of values in the sequence
Max	Returns null	Returns null	Returns the maximum of the non-null values in the sequence	Returns the maximum of all of the values in the sequence
Min	Returns null	Returns null	Returns the minimum of the non-null values in the sequence	Returns the minimum of all of the values in the sequence
Sum	Returns null	Returns null	Returns the sum of the non-null values in the sequence	Returns the sum of all of the values in the sequence

Interacting with type

Shaping data by converting its type from one form to another lets you perform additional tasks, such as creating specific output views. For example, it's common to convert data to a string type so that it's possible to use the string methods to manipulate the appearance of the data in certain ways, such as to make the data more aesthetically pleasing to the viewer.

The only types that you can convert or test are those that map to an Entity Framework type. This functionality works at the conceptual level, rather than at the data source, as does some of the other functionality discussed so far. The two common methods for converting and testing data are *Convert()* (primitive types) and *OfType()* (entity types). When working with C#, you can also use the *is()* and *as()* methods.

Tip You can find information about primitive type mapping at *http://msdn.microsoft.com/library/ee382832.aspx*. Entity type mapping information appears at *http://msdn.microsoft.com/library/ee382837.aspx*. Even though the documentation doesn't specifically mention it, you can also use the *OfType()* method with complex types, which are described at *http://msdn.microsoft.com/library/ee382831.aspx*. When working with a DBMS other than SQL Server, you need to find the mapping for that DBMS. For example, the documentation for MySQL appears at *http://www.devart.com/dotconnect/mysql/docs/DataTypeMapping.html*.

Paging the output

Paging methods sort the data by interacting with the rows out of order or shape the data by removing some rows entirely. The output you receive depends on the way in which you use the paging methods in your code. The paging methods are *ElementAt()*, *First()*, *FirstOrDefault()*, *Last()*, *LastOrDefault()*, *Single()*, *Skip()*, *Take()*, and *TakeWhile()*. If you try to use a paging method on a sequence that doesn't contain any entries or contains all null values, the result is null.

Note Not all overrides of all of the paging methods are supported, because there isn't any way to map them to a function at the data source. The functionality you receive from the paging methods depends on the capabilities of the DBMS you work with. Some DBMSs will return a default value for some methods, and this value is always converted to an Entity Framework primitive type result or a reference type with a null default. Unless your ADO.NET provider fully documents the Entity Framework paging method functionality supported, you'll need to test this functionality as part of your application (realizing that it may not work at all).

Summarizing the LINQ operators

LINQ (and by extension LINQ to Entities) supports a number of operators that you access as methods. The following list provides a description of each of these methods; you can use it to determine which to use to perform a specific task:

- ***Aggregate()*** Applies an accumulator function over the elements of a sequence. For example, you might choose to concatenate the individual strings of a series of records together. You can read more about this method at *http://msdn.microsoft.com/library/bb548651.aspx*.
- ***All()*** Determines whether all of the elements in a sequence satisfy a particular condition. You can read more about this method at *http://msdn.microsoft.com/library/bb548541.aspx*.
- ***Any()*** Determines whether a sequence contains any elements. You can read more about this method at *http://msdn.microsoft.com/library/bb337697.aspx*.
- ***Average()*** Computes the average of the elements found in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb354760.aspx*.
- ***Concat()*** Adds (concatenates) one sequence to another, so that you end up with a single sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb302894.aspx*.
- ***Contains()*** Looks for the specified element in the specified sequence using the default equality comparator. You can read more about this method at *http://msdn.microsoft.com/library/bb352880.aspx*.
- ***Convert()*** Changes the base type of an element into another base type. You can read more about this method at *http://msdn.microsoft.com/library/system.convert.aspx*.

- ***Count()*** Obtains the number of elements in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb338038.aspx*. (See the *LongCount()* method when you want to count a large number of elements.)
- ***DefaultIfEmpty()*** Returns the sequence when there are elements to return. Otherwise, this method returns the default value for the specified sequence, which will likely be an empty or null value. You can read more about this method at *http://msdn.microsoft.com/library/bb360179.aspx*.
- ***Distinct()*** Returns only the unique elements from a sequence. When two elements have the same value, returns just one of the two elements. You can read more about this method at *http://msdn.microsoft.com/library/bb348436.aspx*.
- ***ElementAt()*** Returns the element found at the specified index. You can read more about this method at *http://msdn.microsoft.com/library/bb299233.aspx*.
- ***EqualAll()*** Determines whether two sequences are precisely equal, which means that they must have the same members appearing in the same order. This operator isn't documented as a standard LINQ operator, so Microsoft may restrict its use. You can read more about this method at *http://msdn.microsoft.com/vstudio/bb737910.aspx*.
- ***Except()*** Creates a sequence that contains the elements that don't match between two sequences. The comparison is made using the default comparer. You can read more about this method at *http://msdn.microsoft.com/library/bb300779.aspx*.
- ***First()*** Returns the first element in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb291976.aspx*.
- ***FirstOrDefault()*** Returns the first element in a sequence or a default element when no elements exist. You can read more about this method at *http://msdn.microsoft.com/library/bb340482.aspx*.
- ***GroupBy()*** Places the elements in a sequence in groups using the specified key. You can read more about this method at *http://msdn.microsoft.com/library/bb534501.aspx*.
- ***GroupJoin()*** Combines and groups two separate sequences into a single sequence using a common attribute or property. The resulting groups are based upon the same type of expression used to group a single sequence using the *Group()* method. You can read more about this method at *http://msdn.microsoft.com/library/bb534675.aspx*.
- ***Intersect()*** Produces the set intersection of two sequences by using the default comparator. You can read more about this method at *http://msdn.microsoft.com/library/bb460136.aspx*.
- ***Join()*** Combines two separate sequences into a single sequence using a common attribute or property. You can read more about this method at *http://msdn.microsoft.com/library/bb534675.aspx*.
- ***Last()*** Returns the last element in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb358775.aspx*.

- ***LastOrDefault()*** Returns the last element in a sequence or a default element when no elements exist. You can read more about this method at *http://msdn.microsoft.com/library/bb301849.aspx*.
- ***LongCount()*** Obtains the number of elements in a sequence and returns that value as a 64-bit number. You use this version of *Count()* when the number of elements is high and you want to avoid a potential overflow condition. You can read more about this method at *http://msdn.microsoft.com/library/bb353539.aspx*.
- ***Max()*** Determines which element contains the maximum value in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb335614.aspx*.
- ***Min()*** Determines which element contains the minimum value in a sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb298087.aspx*.
- ***OfType()*** Determines whether an element is of a specific type. You can read more about this method at *http://msdn.microsoft.com/library/bb360913.aspx*.
- ***OrderBy()*** Sorts the elements of a sequence in ascending order using the specified key. You can read more about this method at *http://msdn.microsoft.com/library/bb534966.aspx*.
- ***OrderByDescending()*** Sorts the elements of a sequence in descending order using the specified key. You can read more about this method at *http://msdn.microsoft.com/library/bb534855.aspx*.
- ***Reverse()*** Inverts the order of the elements in a sequence. The elements aren't sorted—merely reversed in order. You can read more about this method at *http://msdn.microsoft.com/library/bb358497.aspx*.
- ***Select()*** Chooses each element of a sequence and optionally modifies its form. You can read more about this method at *http://msdn.microsoft.com/library/bb548891.aspx*.
- ***SelectMany()*** Chooses each element of a sequence, places it in an *IEnumerable* object, and flattens the entire sequence into a single sequence. You can read more about this method at *http://msdn.microsoft.com/library/bb534336.aspx*.
- ***Single()*** Returns the only element in a sequence that satisfies the specified condition and throws an exception if more than one element that satisfies the condition exists. You can read more about this method at *http://msdn.microsoft.com/library/bb155325.aspx*.
- ***Skip()*** Bypasses (skips) the specified number of elements in a sequence and then returns the elements that remain. You can read more about this method at *http://msdn.microsoft.com/library/bb358985.aspx*.
- ***Sum()*** Adds (sums) the individual values of each element in a sequence to create a total. You can read more about this method at *http://msdn.microsoft.com/library/bb298138.aspx*.
- ***Take()*** Returns the specified number of elements in a sequence and then skips (bypasses) the elements that remain. You can read more about this method at *http://msdn.microsoft.com/library/bb503062.aspx*.

- ***TakeWhile()*** Returns the specified number of elements in a sequence while the specified condition remains true, and then skips (bypasses) the elements that remain. You can read more about this method at *http://msdn.microsoft.com/library/bb534804.aspx*.
- ***ThenBy()*** Performs a subsequent sorting of elements in a sequence in ascending order using the specified key. You must precede this method call with either the *OrderBy()* or *OrderByDecending()* method call. You can read more about this method at *http://msdn.microsoft.com/library/bb534743.aspx*.
- ***ThenByDescending()*** Performs a subsequent sorting of elements in a sequence in descending order using the specified key. You must precede this method call with either the *OrderBy()* or *OrderByDecending()* method call. You can read more about this method at *http://msdn.microsoft.com/library/bb534736.aspx*.
- ***Union()*** Produces the set union of two sequences by using the default comparator. You can read more about this method at *http://msdn.microsoft.com/library/bb341731.aspx*.
- ***Where()*** Filters a sequence based on the criterion you provide in the form of an expression. You can read more about this method at *http://msdn.microsoft.com/library/bb534803.aspx*.

Understanding LINQ compilation

LINQ to Entities compiles the queries you create into something that the *EntityClient* can understand. You've seen one example of this compilation in the "Developing LINQ to Entities queries" section of the chapter in the form of bubble help. You were able to hover the mouse over the *CustomerList* object and see its type.

The following sections look at compilation in another way. These procedures take you through the process of using a query with the debugger. It's interesting to see how the debugger handles the query based on the way you create it. In fact, using the debugger as shown in the following procedures will help you gain a much better understanding of the Entity Framework as a whole because you can trace through the tasks it performs in the background for you.

Following an *IQueryable* sequence

The example shown in the "Developing LINQ to Entities queries" section of the chapter uses the *var* keyword to create the *CustomerList* object. The *var* keyword is also used to create *Customer* and *Purchase*. When using the *var* keyword, you allow the compiler to automatically determine which type to use to satisfy a particular need. However, it's nice to see this process in action.

Simply running the example leaves some questions unanswered. For example, you may wonder how and when *Customer* and *Purchase* are created. Working through the example with the debugger helps you answer these kinds of questions.

Tracing through an *IQueryable* example

1. Open the *ModelFirst* example that you worked with in the "Developing LINQ to Entities queries" section of the chapter.

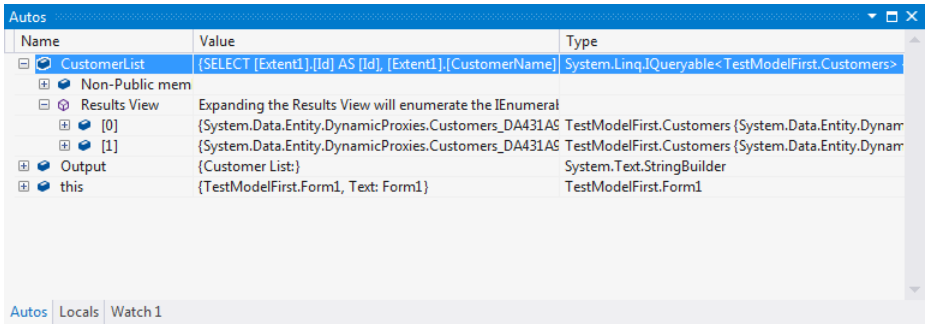
2. Place a breakpoint at the *foreach* line so that it looks like this:

```
// Process each customer in the list.
StringBuilder Output =
    new StringBuilder("Customer List:");
foreach (var Customer in CustomerList)
{
    // Create a customer entry for each customer.
    Output.Append("\r\n" + Customer.CustomerName +
        " has made purchases on: ");
```

3. Click Start or press F5. The application compiles and runs.

4. Click Query. The debugger stops the application at the *foreach* line. There are some interesting things to see at this point.

5. Choose Debug | Windows | Autos. You'll see the Autos window shown here:



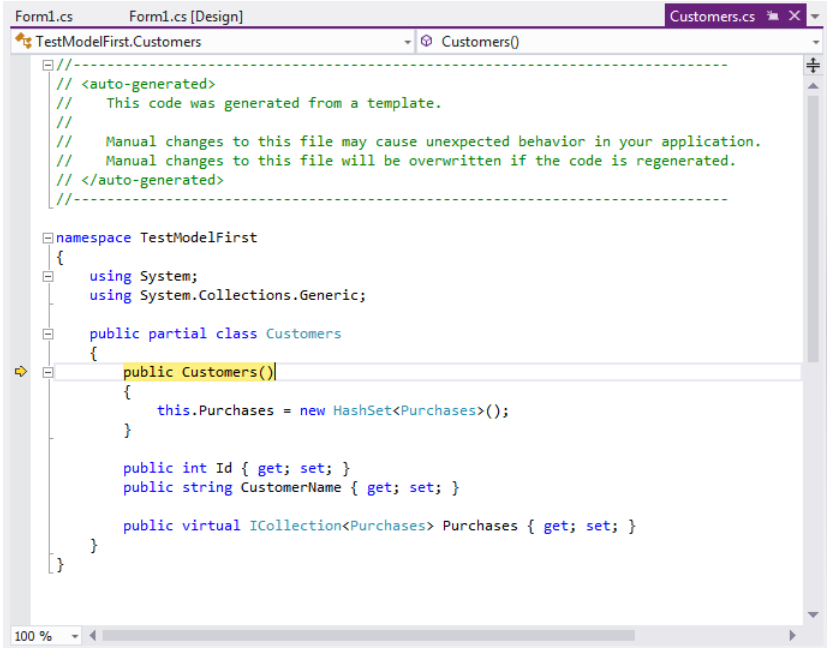
Notice that even though *CustomerList* uses *var* as its type, the actual type is *IQueryable*. The value of *CustomerList* is a form of the query you used.

When you open the Results View, you see that there are two members of type *System.Data.Entity.DynamicProxies*. When working with the Entity Framework, it actually creates a dynamically generated derived type that acts as a proxy for the entity. You can read about these proxies at *http://msdn.microsoft.com/data/jj592886.aspx*. For now, it's important to realize that the *TestModelFirst.Customers* objects don't actually exist.

6. Expanding the Results View has automatically created the customers for you, so click Stop.

7. Perform steps 3 and 4 again to restart the debugger.

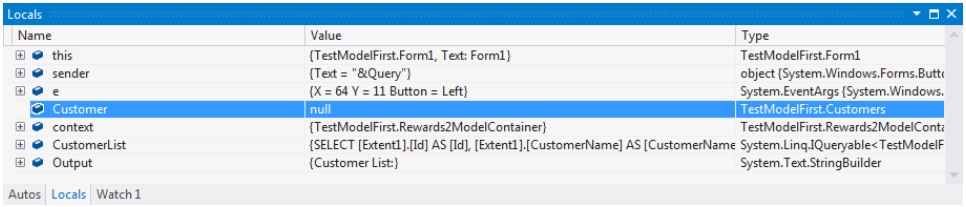
8. Click Step Into or press F11 three times. Visual Studio opens a new file, *Customers.cs*, and places the instruction pointer on the constructor for the *Customers* class, as shown here:



Here, the application is actually creating a *Customers* object. This object includes *Purchases*, as shown.

9. Click Step Into or press F11 four times. The debugger takes you back to the original file and highlights the *in* part of the *foreach* loop, where it verifies that there is another item to process.

10. Click Step Into or press F11. The debugger highlights the *var Customer* part of the *foreach* loop. Choose Debug | Windows | Locals. You'll see the Locals window, as shown here:



Notice that *Customer* is still null. However, the data type shows that *var Customer* creates a *TestModelFirst*.Customers type. The compiler has automatically chosen the correct type for the variable.

11. Click Step Into or press F11. The value of *Customer* changes to a *System.Data.Entity.DynamicProxies* entry. The type is correct for the kind of information presented, and you see the individual values for *Customer* when you click the plus sign next to it.

12. Click Step Into or press F11 six times. The instruction pointer will end up at the *Output.Append()* line. Notice that the application doesn't create the *Purchase* object as it did the *Customer* object. That's because the *Purchase* object already exists as part of the *Customer* object.

13. Click Step Into or press F11 enough times to take the instruction pointer back to the *in* part of the *foreach* loop. When you click Step Into or press F11 one more time, the debugger reopens *Customers.cs*, and you start the process of creating a *Customers* object again, as described in step 9. You can follow this process at least twice if you created the records described in previous chapters.

14. Click Stop to end the debugging session. At this point, you know that working with the Entity Framework with *IQueryable* means creating objects on demand.

Following a *List* sequence

Working with *IQueryable* produces one result. However, converting the query to a *List* and then processing that *List* produces another. It's interesting to modify the code slightly to see what happens when you use a List to interact with a LINQ to Entities query. The following procedure does just that.

Tracing through a *List* example

1. Modify the query in the *ModelFirst* example so that it looks like this:

```
// Obtain the customer list in list form.
List<Customers> CustomerList =
    (from cust in context.Customers
     select cust).ToList<Customers>();
```

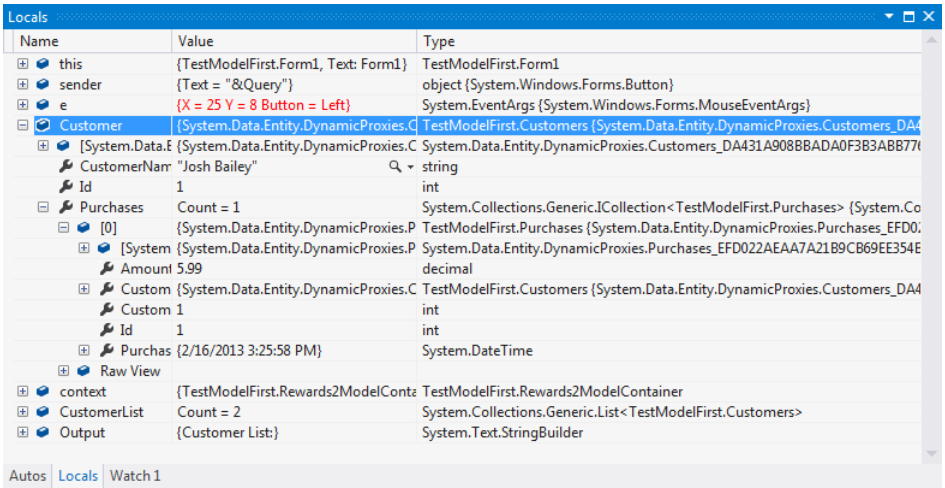
   The result of the query is the same. The only difference is that the output is converted to a *List*.

2. Click Start or press F5. The application compiles and runs.

3. Click Query. The debugger stops the application at the *foreach* line.

4. Click Step Into or press F11 four times. You end up at the opening curly brace for the *foreach* loop. Notice that the debugger didn't open Customers.cs or interact with the constructor in that file. That's because the act of converting the query output to a *List* automatically retrieves the data from the database.

5. Choose Debug | Windows | Locals. You'll see the Locals window shown here:



Notice that, even though the *CustomerList* type is not *System.Collections.Generic.List<TestModelFirst.Customers>*, the *Customer* object hasn't changed from before. It's still of type *TestModelFirst.Customers* and contains a *System.Data.Enty.DynamicProxies* value. The only change that using a *List* creates is the fact that the data entries are retrieved immediately, rather than as needed. That said, using a *List* could save time when working with larger data-sets. You could always create a thread for the data retrieval process so the user can continue working in the foreground.

6. Click Stop to stop the debugger.

Using entity and database functions

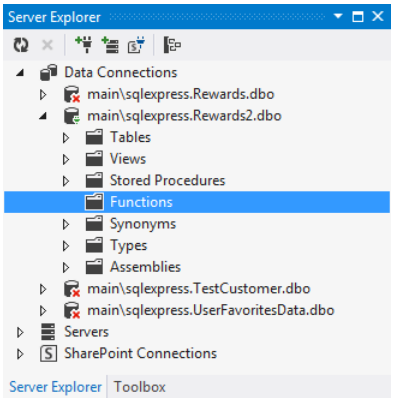
Functions are an important part of modern database applications. You use them to perform a variety of tasks, such as finding the average value of a customer's purchases. Creating and using functions need not be a grueling task. The following sections describe how to create and use functions with the Entity Framework. You can find this example in the \Microsoft Press\Entity Framework Development Step by Step\Chapter 06\ModelFirst (Function) folder of the downloadable source code.

Creating the function

Before you can use a function, you must create it. The following procedure demonstrates one technique for creating functions in SQL Server without leaving the Visual Studio IDE. The procedure relies on the *ModelFirst* example you created in the "Creating a model-first example" section of Chapter 3.

Defining a function using Visual Studio

1. Copy the *ModelFirst* example you created in Chapter 3 to a new folder and use this new copy for this example (rather than the copy you created in Chapter 3).
2. Choose View | Server Explorer. You'll see the Server Explorer window shown here:



3. Open the Rewards2 connection.
4. Right-click the Functions folder and choose Add New | Table-Based Function. Visual Studio opens a new SQL file for you that contains a basic template for creating table-based functions.
5. Type the following code into the file:

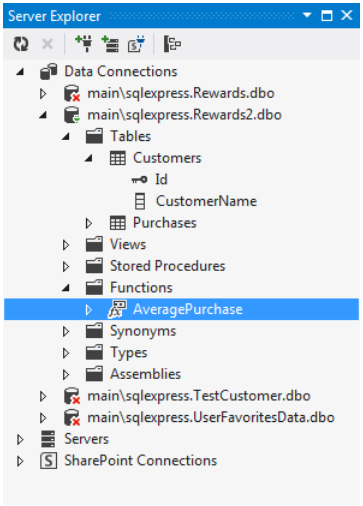
```
USE [Rewards2]
GO

CREATE FUNCTION [dbo].[AveragePurchase]
(
   @CustomerId int
)
RETURNS DECIMAL(3,2)
AS
BEGIN
   DECLARE @Average DECIMAL(3,2)
   SELECT @Average = avg(Amount)
      FROM Purchases
      WHERE CustomersId = @CustomerId;
   RETURN @Average
END
```

This function begins by selecting the appropriate database for modification. It then creates a function named *AveragePurchase*, which accepts a single input, *CustomerId*. The function creates a variable, *@Average*, of type *DECIMAL*, and uses it as part of an SQL statement that selects the average of the purchases contained in *Amount* from the *Purchases* table, where the *CustomerId* value matches the *@CustomerId* input. The result is the average purchase amount for a single customer.

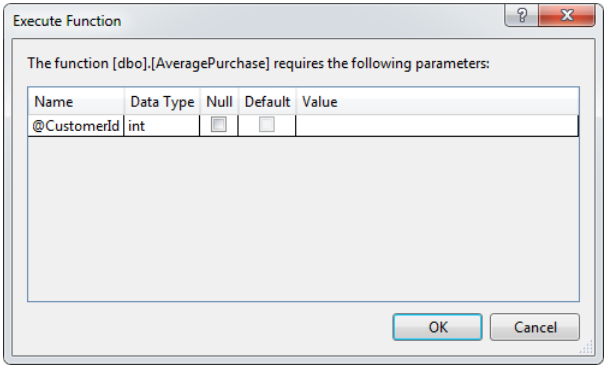
6. Right-click anywhere in the code window and choose Execute from the context menu. You'll see the Connect To Server dialog box.

7. Provide the required credentials and click Connect. Visual Studio will execute the command for you. You should see "Command(s) completed successfully." on the Message tab that appears when you execute the command.

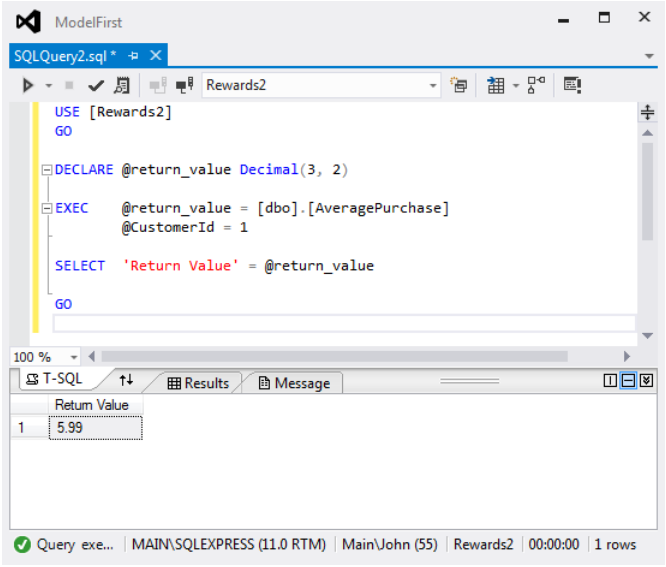
8. Right-click the Rewards2 entry in Server Explorer and choose Refresh from the context menu. You'll see the new function appear in the Functions folder, as shown here:



9. Right-click *AveragePurchase* and choose *Execute* from the context menu. You'll see an Execute Function dialog box like the one shown here, telling you the function requires an input value to execute:



10. Type **1** in the Value field for *@CustomerId* and click OK. Visual Studio automatically creates a new query and executes it. You'll see the output shown here:



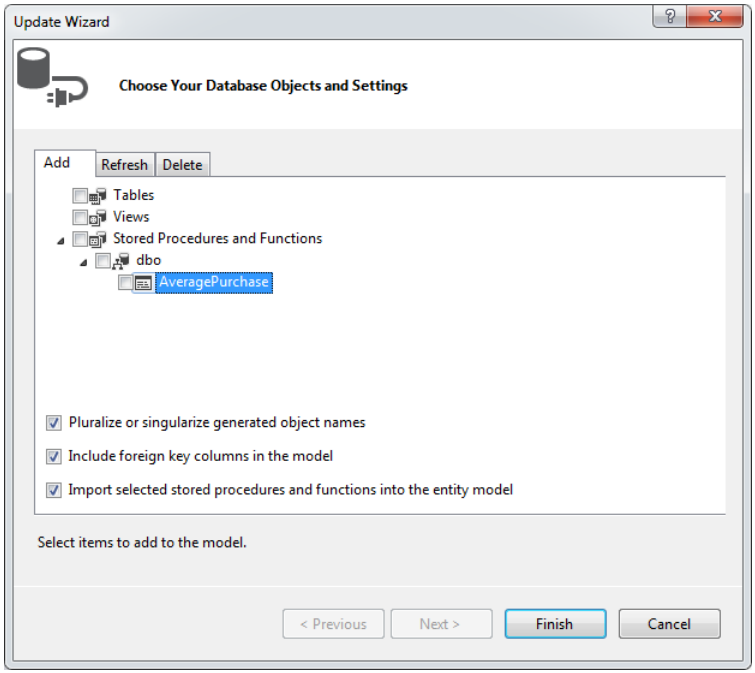
11. Close the SQL file without saving it. The test shows that the query works.

Accessing the function

At this point, you have a database function you can use. You know it works because you tested it. Of course, you have to figure out how to access the function from your code. The following procedure shows how to access the function from within your application.

Tracing through a *List* example

1. Open the Rewards2Model.EDMX file by double-clicking its entry in Solution Explorer.
2. Right-click in any clear area of the designer and choose Update Model From Database from the context menu. You'll see the Update Wizard dialog box shown here:



3. Check AveragePurchase and click Finish. It seems as if nothing has happened to your diagram, but the .EDMX file does indeed include a change.

4. Open the Form1.cs file. Add this *using* statement to the beginning of the file:

```
using System.Data.Objects.DataClasses;
```

5. Add this function to the file:

```
[EdmFunction("Rewards2Model.Store", "AveragePurchase")]
public static decimal? AveragePurchase(Int32 CustomerId)
{
   throw new NotSupportedException("Direct calls are not supported.");
}
```

This function requires a little explanation. The *[EdmFunction()]* attribute tells the compiler to look into the .EDMX file in the requested store, which is Rewards2Model.Store in this case, for a function named *AveragePurchase*. You added this entry during the update, even though it doesn't show up in the designer.

The function itself requires an odd format. For one thing, it's a static function, and the return type is *decimal*. Notice the question mark (?) behind the type declaration. You must include it or the function won't work. The function name comes next, along with any arguments the function requires. The only content for the function is the exception shown. The function actually executes at the database.

6. Add a new button to *Form1*. Name the button *btnAverage* and set its *Text* property to *&Average*.

7. Double-click *btnAverage* to create a new click event handler.

8. Type the following code for the *btnAverage_Click()* event handler:

```
private void btnAverage_Click(object sender, EventArgs e)
{
   // Create the context.
   Rewards2ModelContainer context = new Rewards2ModelContainer();

   // Make the query.
   var CustomerList =
      from cust in context.Customers
      select new
      {
         Name = cust.CustomerName,
         Average = AveragePurchase(cust.Id)
      };

   // Create a string to hold the result.
   StringBuilder Output = new StringBuilder();

   // Parse the result.
   foreach (var CustEntry in CustomerList)
      Output.Append(
         CustEntry.Name + " makes an average purchase of "
         + CustEntry.Average + ".\r\n");

   // Display the result on screen.
   MessageBox.Show(Output.ToString());
}
```

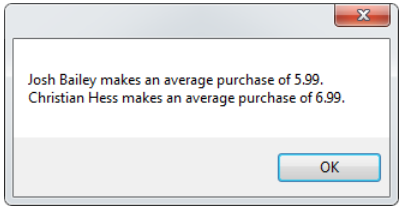
The code begins by creating a context. It then creates a LINQ to Entities query based on that context. Notice that the *select* part of the query is different. It creates a new object that contains two entries: *Name* and *Average*. The *Name* entry is directly obtained from *cust.CustomerName*. However, the *Average* entry is actually a call to the *AveragePurchase()* function you created in the database in the "Creating the function" section. What you end up with is a structure-like *IQueryable* object. (Tracing through this example in the debugger is educational, and you should give it a try.)

After the application obtains the names and averages, it creates a string from them using a *foreach* loop. Notice that you access the entries as properties. *CustEntry* is actually an anonymous type. The code ends by displaying the output in a message box.

Note This is an example of an application where you must use *var* instead of either *IQueryable* or *IEnumerable*. The problem is that you're working with an anonymous type—a type that isn't known at design time.

9. Click Start or press F5. The application compiles and runs.

10. Click Average. You'll see the output shown here:



Getting started with the Entity Framework

This chapter has introduced you to LINQ to Entities, which provides a method of querying a database using a simple and straightforward query language. The most important idea to take away from this chapter is that LINQ to Entities makes it possible to focus on the information you need to work with, rather than the method used to obtain it. In order to define what information you need, a declarative language uses a set of keywords and operators that make it possible to tell the compiler what you want. LINQ to Entities queries are compiled into a form that the .NET Framework understands. So, there isn't any hocus-pocus going on—LINQ to Entities simply makes it possible for you to get your work done faster and with fewer errors.

The chapter contains a number of examples. What you need to do at this point is play with those examples to determine how they work. If necessary, single-step through the code using the debugger to determine precisely how the queries work. Once you understand the queries as they appear in the chapter, make changes to them to see how different operators and keywords affect the output. The best way to gain an appreciation of how LINQ to Entities works is to play with it. Spend some time mixing and matching items until you gain a clear understanding of how each item works.

Chapter 7 moves on to another way of interacting with data, using Entity SQL. In this chapter, you gain an in-depth view of working with Entity SQL to perform specific tasks. As in Chapter 6, you start with a basic tutorial of how Entity SQL works, and then move on to examples that demonstrate how to use it. When you finish Chapter 7, you'll be able to compare LINQ to Entities with Entity SQL to determine the strengths and weakness of each approach. You'll also have a better idea of which technology you prefer to use to address a particular need.

Chapter 6 quick reference

To	Do this
Access a non–SQL Server database using LINQ to Entities	Obtain the required database-specific provider to use with ADO.NET.
Create a basic LINQ to Entities query	Combine the *from*, *in*, and *select* keywords to create an expression, and then place the output from this expression into a variable. For example, *var CustomerList = from cust in context.Customers select cust* obtains a list of all of the customers found in the *Customers* table of the specified context named *context*.
Specify that LINQ group the return values in a certain way	Use the *group* keyword to specify that you want grouping and the *by* keyword to define which field or expression to use to perform the grouping task. Place the result of the grouping into a variable by using the *into* keyword.
Specify that LINQ sort the return values in a certain way	Use the *orderby* keyword to specify that you want the output sorted and include a field or expression to use to perform the sorting task. Control the order of the sort using the *ascending* or *descending* keyword.
Output a result set using an in-memory presentation that provides performance benefits during enumeration	Create an output object based on *IEnumerable*.
Output a result set using a remote presentation that provides flexibility	Create an output object based on *IQueryable*.
Project specific output values from the query	Use the *Select()* or *SelectMany()* methods.
Filter the output to remove undesirable elements	Use the *Where()* method.
Join two data sources that lack a navigable property	Use the *Join()* or *GroupJoin()* method to create an inner, group, or left-outer join.
Create a result set that exhibits one or more specific properties	Use the set-related methods: *All()*, *Any()*, *Concat()*, *Contains()*, *DefaultIfEmpty()*, *Distinct()*, *EqualAll()*, *Except()*, *Intersect()*, and *Union()*.
Change the order in which the rows in a result set appear	Use the ordering-related methods: *OrderBy()*, *OrderByDescending()*, *ThenBy()*, *ThenByDescending()*, and *Reverse()*.
Define groups of rows containing the same attribute	Use the *GroupBy()* method.
Define new views of existing data by combining rows	Use the aggregation-related methods: *Aggregate()*, *Average()*, *Count()*, *LongCount()*, *Max()*, *Min()*, and *Sum()*.
Perform type conversion and testing	Use the *Convert()* (primitive types) and *OfType()* (entity types). When working with C#, you can also use the *is()* and *as()* methods.
Access the rows out of order or remove some rows from the sequence depending on position	Use one of the paging methods: *ElementAt()*, *First()*, *FirstOrDefault()*, *Last()*, *LastOrDefault()*, *Single()*, *Skip()*, *Take()*, or *TakeWhile()*.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

About the Author

JOHN PAUL MUELLER is a freelance author and technical editor. He has writing in his blood, having produced 92 books and over 300 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books include Windows command-line references, books on HTML5 and JavaScript, several books on C#, and an IronPython programmer's guide. His technical-editing skills have helped more than 65 authors refine the content of their manuscripts. John has provided technical-editing services to both *Data Based Advisor* and *Coast Computer* magazines. He's also contributed articles to magazines such as *Software Quality Connection*, *Mendix.com*, *DevSource*, *InformIT*, *SQL Server Professional*, *Visual C++ Developer*, *Hard Core Visual Basic*, *asp.netPRO*, *Software Test and Performance*, and *Visual Basic Developer*. Be sure to read John's blog at *http://blog.johnmuellerbooks.com/*.

When John isn't working at the computer, you can find him outside in the garden, cutting wood, or generally enjoying nature. John also likes making wine and knitting. When not occupied with anything else, he makes glycerin soap and candles, which comes in handy for gift baskets. You can reach John on the Internet at *John@JohnMuellerBooks.com*. John is also setting up a site at *http://www.johnmuellerbooks.com/*. Feel free to take a look and make suggestions on how he can improve it.

Now that you've read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!