Pro LINQ Object Relational Mapping with C# 2008

ISBN-13 (pbk): 978-1-59059-965-5

ISBN-10 (pbk): 1-59059-965-9

ISBN-13 (electronic): 978-1-4302-0597-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Fabio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Sharon Wilkey
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Carol Burbo
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

PART 1

Object-Relational Mapping Concepts

CHAPTER 1

Getting Started with Object-Relational Mapping

In the introduction, I stated that the purpose of this book is to explore object-relational mapping (ORM) by examining the new tools, LINQ to SQL and EF, as well as tried-and-true design patterns. Unfortunately, to become a "professional" at ORM development, you have to start with the basics. This chapter introduces you, largely in a technology-independent manner, to some of the essential concepts of ORM. In addition to a basic examination of the "what and why" of ORM, you will also be exploring the qualities that make up a good ORM tool and learning whether LINQ to SQL and EF make use of them.

Introduction to Object-Relational Mapping

Developing software is a complicated task. Sure, developing software has come a long way from the assembler days, and we have all sorts of managed, interpreted, Fisher-Price languages and tools to make things easier for us; even so, things don't always work as intended. This is especially evident when a software application has to connect to a relational database management system (RDBMS). Anyone with experience in this area knows it is rare in modern enterprise architecture for a piece of software to live in a silo, without needing to connect to some sort of database.

Although some might disagree, relational database systems are really considered the lifeline of every enterprise application and, in many cases, of the enterprise itself. These remarkable systems store information in logical tables containing rows and columns, allowing data access and manipulation through Structured Query Language (SQL) calls and data manipulation languages (DMLs). Relational databases are unique in the enterprise because they form the foundation from which all applications are born. In addition, unlike other software applications, databases are often shared across many functional areas of a business. One question that I've been asked in the past is this: if databases have all the data, why don't we just write all our software in the database? After I controlled my laughter, I began to really think about this question. From the eyes of a business *user*, it makes perfect sense to have a single layer in the architecture of the system, rather than multiple layers. You would have fewer servers and moving parts—and to paraphrase Mark Twain, as long as you watch that basket closely, it's fine to put all your eggs in a single basket. It makes perfect sense.

But wait a second. That sounds a lot like a mainframe: a single monolithic environment in which you write procedural code to access data and use a nonintuitive user interface (UI) for interacting with that data. Now don't get me wrong—the mainframe has its place in the world (yes, still), but not if you plan to write distributed applications, with rich user interfaces (that is, web, thick client, mobile, and so forth), that are easy to customize and adapt, in a rapid application development environment; these aspects instead require an object-oriented language such as C#, VB.NET, Java, or C++.

If you've decided that you're not going to write an entire application in Transact-SQL (T-SQL), and you've decided to use an object-oriented programming language, what are your next steps? Obviously, you need to go through some sort of process to gather requirements, create a design, develop the software, and test (some people unwisely skip this step). However, during the design phase, how do you plan out your data access layer? Will you use stored procedures for your create, read, update, and delete (CRUD) transactions? Maybe you have built a custom data access layer based on ADO.NET and COM+, or perhaps you have purchased some widget to do this or that. In my experience at Microsoft shops, ORM rarely comes up in the discussion. Why would it? ORM fundamentally goes against the direction that Microsoft pursued for years. Although many Microsoft shops have turned to third-party tools for ORM support, the norm has always been to use `DataSets` and ADO.NET objects. True object-oriented programming techniques, like business objects, were hardly ever discussed, and when they were, they were discussed only as an offshoot of ADO.NET. I suppose you could say the de facto stance of Microsoft and most Microsoft developers has always been that the power of the `DataSet` and `DataTable` was good enough for any enterprise application and any discerning developer.

What Is ORM?

ORM is an automated way of connecting an object model, sometimes referred to as a *domain model* (more on this in the coming chapters), to a relational database by using metadata as the descriptor of the object and data.

■ **Note** I use the word *automated* in the sense that the ORM tool that you are using is neither homegrown nor a manual process of connecting objects to a database. Most people with some basic knowledge of ADO.NET can create a data access layer and populate a business object. In this context, an ORM tool is a third-party tool (for example, LINQ to SQL) that provides you with commercial off-the-shelf mapping functionality.

According to Wikipedia, "Object-relational mapping (a.k.a. O/RM, ORM, and O/R mapping) is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages." Frankly, this definition is good enough for me because it is simple enough for everyone to understand and detailed enough to tell the full story. Over the coming chapters, the semantics of ORM are further refined, but this definition is a good place to start.

Benefits of ORM

It is important to understand that there are many benefits to using ORM rather than other data access techniques. These benefits will become more evident as you work through examples, but the following are ones that most stick out in my mind. First, ORM automates the object-to-table and table-to-object conversion, which simplifies development. This simplified development leads to quicker time to market and reduced development and maintenance costs. Next, ORM requires less code as compared to embedded SQL, handwritten stored procedures, or any other interface calls with relational databases. Same functionality, less code—this one is a no-brainer. Last but not least, ORM provides transparent caching of objects on the client (that is, the application tier), thereby improving system performance. A good ORM is a highly optimized solution that will make your application faster and easier to support.

Those points are important, but let's talk about a real-world situation that I have seen at various companies. Company X has developed a piece of software for a dog food producer and has followed the mantra that stored procedures are the fastest solution, and all CRUD operations should be handled by using stored procedures. The developers at this company have followed this approach to the point that they have individual stored procedures for each CRUD transaction—more than 3,000 stored procedures in all. This approach is a common scenario in the .NET world, with ADO.NET and SQL Server. This software has ballooned so much over the past couple of years because of customizations, a lack of standardization, and novice developers that development costs have doubled.

■ **Note** Writing stored procedures does not equate to bad design. On the contrary, stored procedures if managed correctly are an excellent alternative to dynamic SQL. If you use a code generator to create your CRUD stored procedures, or if you develop them by hand and have good oversight, in many cases you will be able to use them in conjunction with an ORM tool.

Company X has a major dilemma now because it has an opportunity to sell its software to a cat food producer, but the software needs to change to meet the business needs of the cat food company. With the increased development costs, Company X won't be making enough money to justify this deal. But company officials like the idea of selling their software, so they hire a consulting company to help reduce their overhead. The consulting company wants to use an ORM tool to help improve the situation. This consulting company builds an object model to represent the business and to optimize and normalize the company database. In short order, Company X is able to eliminate the thousands of stored procedures and instead use "automagic" mapping features to generate its SQL. Development costs go down, profits go up, and everyone is happy. Clearly this is an oversimplified example, but you can change Company X's name to that of any number of organizations around the world. The point is simple: ORM tools can't make coffee, but they can provide a proven method for handling database access from object-oriented code.

Qualities of a Good ORM Tool

I have often been asked what criteria I like to use when evaluating an ORM tool. Because the primary focus of this text is VS 2008, I've decided to outline all the features I look for, and then discuss if and how LINQ to SQL and EF implement them. In Chapter 12, I present some non-Microsoft commercial ORM tools and use the items in this list to help evaluate their usability.

Object-to-database mapping: This is the single most important aspect of an ORM tool. An ORM tool must have the ability to map business objects to back-end database tables via some sort of metadata mapping. This is fundamentally the core of object-relational mapping (and yes, LINQ to SQL and EF support mapping business objects to database tables with metadata).

Object caching: As the name indicates, this functionality enables object/data caching to improve performance in the persistence layer. Although query and object caching are available in LINQ to SQL and EF, both require some additional code to take advantage of this functionality. You will examine this topic throughout the text as you look at how LINQ to SQL and EF control object and state management. At this time, it is important that you understand only that both tools support caching in some form or another.

GUI mapping: Like so many other topics in the IT world, this is a debated topic. I'm of the mind that software with a graphical user interface (GUI) is a good thing, and the simpler the interface, the better. However, there are still purists in the ORM world who say mappings should be done by hand because this is the only way to ensure that the fine-grained objects are connected correctly. I believe that if an ORM tool has everything else you need, yet no GUI, you should still use it. If the GUI is included, consider it the icing on the cake. After all, if the framework is in place, you can always write your own GUI. In the case of LINQ to SQL and EF, GUI designers are provided with Visual Studio 2008.

Multiple database-platform support: This is pretty self-explanatory: a decent ORM offers portability from one RDBMS provider to another. At the time that I'm writing this, though, I am sad to say that LINQ to SQL supports only SQL Server 2000 and up. However, this is the first version of this tool. EF, on the other hand, uses the provider framework and supposedly will support multiple database platforms. It's unclear whether these providers will be available for release to manufacturing (RTM), but again the provider model is part of the strategy. Although I consider this a critical piece of functionality for an ORM tool, providing examples in anything other than SQL Server is outside the scope of this book.

Dynamic querying: Another important aspect of ORM, and the bane of database administrators (DBAs) everywhere, is dynamic query support. This means that the ORM tool offers projections and classless querying, or dynamic SQL based on user input. This functionality is supported natively in LINQ to SQL and EF, allowing users to specify a filter or criteria, and the framework automagically generates the query based on the input.

Lazy loading: The purpose of lazy loading is to optimize memory utilization of database servers by prioritizing components that need to be loaded into memory when a program is started. Chapter 2 details the lazy-loading pattern, so for now just know that lazy loading typically improves performance. LINQ to SQL has a built-in functionality that allows you to specify whether properties on entities should be prefetched or lazy-loaded on first access. This is actually built right into the VS 2008 designer as a property on the entity. EF also supports lazy loading by default; however, there are a few caveats with the object context and concurrency, which you will look at in more depth in upcoming chapters.

Nonintrusive persistence: This is an important one, and it is discussed in depth in Chapter 2. Nonintrusive persistence means that there is no need to extend or inherit from any function, class, interface, or anything else provider-specific. LINQ to SQL supports this concept, because you can use a custom class that does not have any provider-specific technology, and still have it participate in the ORM process. It's not as black-and-white with the Entity Framework because EF does not support nonintrusive persistence. EF does support the use of the `IPOCO` pattern, which you will explore later, but natively you are required to inherit from and extend EF-specific technology to use this ORM.

Code generation: This is another gray area of ORM. The purists will insist that there is no place for code generation in ORM, because a well-thought-out object model should be coded by hand. It should, in fact, be based on the conceptual model of the business domain, not on the metadata of the database. There is room for code generation when using an abstract data object layer, which extends your object model, but that is a different situation that I discuss in the next chapter. I think that code generation can be useful when working on a project in which the database schema is static, and the customer understands the ramifications of using this approach (see the Bottom Up approach). However, even though LINQ to SQL and EF support code generation, I'm not a big proponent of it. I think that the semantics of the object model get lost along the way and the database schema becomes the focus, thus making the application rigid and inflexible. Nonetheless, code generation is an important aspect of these tools, so the text does present some examples.

Multiple object-oriented framework support: This is blue sky. You would be hard-pressed to find an ORM product offering compatibility with multiple object-oriented languages and development environments. I'm not talking about Visual Basic (VB) and C#; rather I'm referencing .NET and Java. This may sound far-fetched, but at some point in the future I envision the persistence layer becoming language agnostic. You say CLR, I say JVM . . . can't we all just get along? This ranks very low on the determining factors for choosing an ORM vendor; however, I like to keep it on the list just to keep everyone on their toes.

Stored procedure support: The object purists are going to read this and say that stored procedures serve no purpose in an ORM tool. Why on earth would you defile your decoupled model and data layer with the integration of stored procedures? The fact of the matter is, in many large organizations you can't get away with using dynamic SQL. The DBA group may have had a bad experience with ORM, or they may enjoy holding all the cards, or may just not like what you have to say. Additionally, because Microsoft has been pushing stored procedures on developers and DBAs for years, it may be difficult for you to change the stored procedure culture overnight. Along with the possibility that the DBA group is standing in your path, at times stored procedures are really the only viable option because of performance problems with long-running or complex queries in the ORM tool (for example, reporting). Regardless of the situation, you're not out of luck because many ORM tools, including LINQ to SQL and EF, support stored procedures.

Miscellaneous: I once worked for a guy who said to never put *miscellaneous* in a list because it showed a lack of completeness. Well, I think miscellaneous is a good way to describe the following items that aren't worth a subheading, but still need to be mentioned for completeness. The miscellaneous criteria are as follows: price, ease of use, documentation, market penetration, performance, and support. I would include all these criteria in an ORM tool analysis, but they are not really relevant to the heart of this text.

The most important thing for you to remember when choosing and using an ORM tool is that it is not going to solve world hunger. It is important that you familiarize yourself with ORM and the tool before making any significant changes to your architecture. ORM tools can increase productivity and decrease time to market, but they can also do the opposite if not fully understood by the stakeholders involved.

Impedance Mismatch

It would be utterly irresponsible of me not to include a section about the *impedance mismatch* that occurs between object code and relational databases. This is probably the single most common explanation that people give for using ORM tools. The object-oriented archetype is founded on the principle that applications are built by using a collection of reusable components called objects. On the other hand, the relational database pattern is one in which the database stores data in tabular form. Whereas the database is largely based on a purely mathematical algorithm, the object-oriented model is based on a representation of life and one's surroundings. Therefore, to overcome the disparities between the two paradigms, it is necessary to map the object model to the data model.

Let's look at an example of the paradigm mismatch. In this example, we begin with a simple class model and slowly expand the model to illuminate the mismatch problem. Here we have the start of a retail banking application, with a `Customer` class and an `Account` class. As seen in Figure 1-1, the `Customer` class has one or more `Account` classes, similar to a customer at a bank.

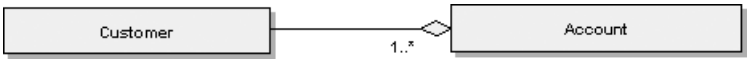

Figure 1-1. *An example of a Customer class with one or more Account classes*

The source code associated with Figure 1-1 resembles the following:

```
public class Customer
{
     private string _firstName;
     private string _lastName;
     private string _fullName;
     private List<Account> _accounts = new List<Account>();
     private int _id;

     // Get and Set Properties for each of our member variables
}

public class Account
{
     private int _id;
     private int _accountNumber;
     private int _customerID;

     // Get and Set Properties for each of our member variables

}
```

The Data Definition Language (DDL) for Figure 1-1 looks similar to this:

```
CREATE TABLE [Customer](
     CustomerID] [int] IDENTITY(1,1) NOT NULL,
     FirstName] [nvarchar](50) NULL,
     LastName] [nvarchar](50) NULL,
     MiddleName] [nvarchar](50) NULL,
     FullName] [nchar](10) NULL,
     CONSTRAINT PK_Customer PRIMARY KEY ([CustomerID])
)

CREATE TABLE [dbo].[Account](
     AccountID] [int] IDENTITY(1,1) NOT NULL,
     AccountNumber] [int] NULL,
     CustomerID] [int] NOT NULL,
       CONSTRAINT PK_Account PRIMARY KEY ([AccountID]),
       CONSTRAINT FK_Account_Customer FOREIGN KEY (CustomerID)
          REFERENCES [Customer]([CustomerID])
)
```

In this scenario, we have a pretty vanilla example: we have one class for one table, and we have a foreign-key relationship of the account table containing the ID of the customer table—if only every database and application were this easy to design. I guess I would probably be out of a job, so maybe it's a good thing that ORM, software engineers, and design patterns are needed.

Let's expand this case so it is based more on a system that you might see in the real world. In the preceding case, you have a `Customer` class that uses strings for the first and last name.

Suppose after speaking with the business, you realize that the banking software will be used in various countries and regions around the world. You know from reading a white paper on internationalization and localization that you are going to need *finer-grained* control of the names in the system because not all cultures use first and last names in the same way. Additionally, after looking at the object model, you realize that it is lacking abstraction, which is really just a nice way of saying that your object model is lacking flexibility and extensibility. Therefore, as illustrated in Figure 1-2, you should refactor your model to add inheritance and abstraction and to make your model a closer representation of the business domain.

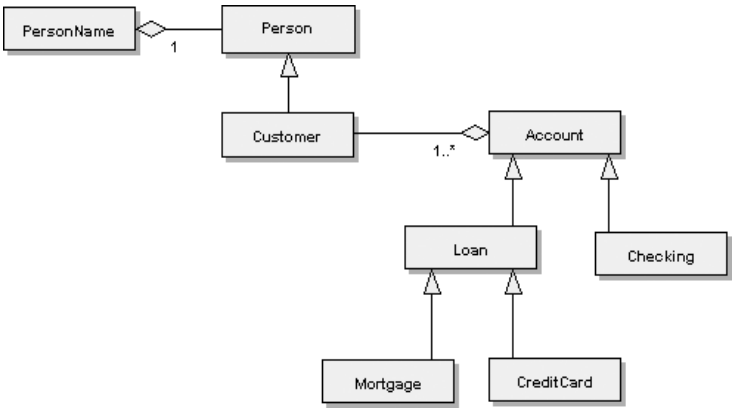

Figure 1-2. *A more realistic representation of an object model for a banking application*

As represented in Figure 1-2, we have expanded our object model to better represent the banking domain. We have modified the `Customer` class so it inherits from a `Person` base class, and we have created a relationship between the `Person` class and the `PersonName` class to account for complex naming situations in foreign countries. In addition, we have expanded the taxonomy of `Account` so that it includes classes for `Loan`, `Checking`, `Mortgage`, and `CreditCard`—all realistic examples of how you would use inheritance for an object model in the banking industry.

In Listing 1-1, you have the diagram from Figure 1-2 enumerated into C#. In this example, you have a better representation of an object model that would be used in the banking industry.

Listing 1-1. *A More Realistic Object Model for the Banking Industry*

```
{
    public int Id {get; set;}
    public PersonName Name {get; set;}
}

public class PersonName
{
    public string FirstName {get;set;}
    public string LastName {get;set;}
    public string FullName{get;set;}
}
```

```
public class Customer:Person
{
    private List<Account> _accounts = new List<Account>();
    public List<Account> Accounts
    {
        get
        {
            return this._accounts;
        }
        set
        {
            this._accounts = value;
        }
    }

}

public class Account
{
    public int Id {get;set;}
    public int AccountNumber{get;set;}
}
```

How does this more-accurate object model in Listing 1-1 relate back to our database schema? If you start with `Person` and `PersonName`, you already can see the mismatch between the data model and object model. In the database schema, it is perfectly acceptable to have a `Customer` table contain all the information for `Person` and `PersonName`. This variance has to do with the difference between fine-grained objects and coarse-grained objects, and the database's inability to handle these common object-oriented relationships and associations.

Let's talk about what I mean here when I say *fine-grained* and *coarse-grained* objects: the analogy comes from any particulate matter (for example, sand) consisting of small or large particles. Like particles, a *coarse-grained object* is one that semantically is large and contains many aspects—for example, the `Account` class in Figure 1-1. Alternatively, *fine-grained objects* are much smaller and detailed, like the refactored version seen in Figure 1-2. The fine-grained approach is almost always preferred when building an object model.

There are several reasons to prefer a group of smaller objects to one giant object, but some of the most important reasons are performance, maintainability, and scalability. Why would you retrieve all the data for a single large object when you can easily retrieve the data you need from a smaller object? Why remote a large object when all you need is a subset of the data? Why make maintenance more complex by putting everything in a single class when you can break it up? These are all questions you should ask the next time you find yourself building an object model.

■ **Tip** In Listing 1-1, I am taking advantage of one of the new C# 3.0 language features, *automatic properties*. As you can see, the code is much cleaner because you do not have to explicitly declare a private field for the property; the compiler does it for you. I use the new language features throughout the text, and they are called out in Tip boxes like this one.

Continuing with the comparison between the object model and the data model, notice that there is significantly more inheritance in the object model in Figure 1-2 as compared to Figure 1-1. This is a common scenario in today's object-oriented world, with applications designed with multiple layers of inheritance for extensibility and abstraction. After all, inheritance is one of the core tenets of object-oriented programming and accordingly is extremely useful when it comes to building flexibility and scalability into your application. Unfortunately, my relational database doesn't come with an inheritance button, and if you think yours does, as they say, I've got a bridge you might be interested in purchasing. Every developer who has worked with a relational database knows that there is no good way to connect multilevel or single-level inheritance relationships in a database. Sure, you can create various relationships and extension tables, and yes, with SQL Server 2005 you can use the Common Language Runtime (CLR); nonetheless, you will still never be able to create an inheritance model in a relational database as cleanly (if at all) as you will in your object model.

Along with inheritance, another core aspect of object-oriented programming rears its head in Figure 1-2; specifically, polymorphism and polymorphic associations are apparent. As you are likely aware, this was bound to happen because anytime you introduce inheritance, there is the possibility of polymorphism. Obviously, this polymorphism depends on your hierarchy and implementation of your concrete classes, but it's safe to say that polymorphism is a possibility when you have inheritance in your object model. A good example of this is seen in the `Customer` class, in which a one-to-many association with the `Account` class exists, and the `Account` class in turn has subclasses of `Loan`, `Checking`, and so forth. Figure 1-2 tells us that the `Customer` object may be associated with any instance of `Account`, including its subtypes at runtime. What this means for you as the developer is that there is a good possibility that you will want to write a query to return instances of the `Account` subclasses. This, of course, is one of the great features of object-oriented programming and one of the drawbacks to relational databases.

As seen in the previous examples, without manipulating the object model with Adapters and other SQL data access code, there is no straightforward way to connect the object-oriented model to a relational database model. The term *Adapters* is taken from *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994) and is defined as the conversion of the interface of a class into another interface that clients expect. Although the examples simply portrayed the inherent mismatch between fine- and coarse-grained objects, inheritance, and polymorphism, I can tell you that encapsulation, abstraction, and the other fundamentals of object-oriented programming don't fare much better. The bottom line: vendors or languages and databases have never addressed the impedance mismatch between object and data, and so ORM continues to thrive.

Object Persistence

Object *persistence* is at the heart of ORM. To quote Obi-Wan: "It surrounds us and penetrates us. It binds the galaxy together." I'm not going to write extensively about persistence because it has been covered exhaustively elsewhere (and if you have picked up this book, I believe you already have a basic understanding of persistence as it relates to object-oriented development). I think it's important, however, to provide a brief refresher and give you my two cents on the subject.

The essence of persistence is the act of saving and restoring data. These objectives are not specific to ORM or object-oriented programming; rather, persistence is a concept that transects

software engineering as a whole. Persistence is apparent every time you turn on your computer, whether you are saving a document, an e-mail, or any other data. The minute you turn on your computer, persistence occurs, from the lowest-level circuit all the way up to the web browser you use to log in to your bank.

In the ORM world, persistence relates to the act of dehydrating and rehydrating (or vice versa) an object model complete with the object's current state. Everyone knows that application data can be persisted to relational database systems for permanent storage, but who wants to always make a round-trip call to the database to retrieve our object data? Additionally, I have already shown that there is an inherent mismatch between an object model and the data model. We need a mechanism to save and restore our object hierarchy, complete with state, in our data access layer: hence, object persistence and ORM.

The primary way to handle object persistence is to utilize object serialization. *Object serialization* is the act of converting a binary object, in its current state, to some sort of data stream that can be used at a later date or moved to an alternate location. The concept isn't new; object persistence has been around a long time. In classic ActiveX Data Objects (ADO), it is possible to serialize a recordset; in Java, you can use Enterprise JavaBeans (EJB) or Plain Old Java Objects (POJOs); and in .NET, you can use ADO.NET or Plain Old CLR Objects (POCOs). Of course, in .NET (and Java), your objects must implement `ISerializable`, allowing the language interpreter to do the serialization work for you. But back to the heart of the matter: when trying to persist an object, serialization is imperative.

This truth was known and embraced by the ORM gurus and developers of yesteryear, and it has carried into the designs of all modern-day ORM tools. Fortunately for us, LINQ to SQL and EF both make good use of object persistence and serialization, making our lives simpler and eliminating the need to spend endless hours designing a persistence mechanism in our data access layer. As you progress deeper into LINQ to SQL and EF, you will explore caching and object persistence in more detail.

Basic ORM Approach

Similar to much of this chapter, the ORM approach is discussed throughout the text; however, I want to set the stage for detailed discussions in the coming chapters. Fundamentally, there are three key approaches when it comes to ORM: the Bottom Up, the Top Down, and the Meet in the Middle approaches. Each approach has its benefits and problems, and no approach should be considered the panacea. It is critical when designing an application to understand that the "one size fits all" mentality never works. Although a chunk of this book looks favorably at the domain-driven design (DDD) model, and the patterns that Martin Fowler and others have produced, the fact is that those patterns and practices are still fallible. A more-holistic view of the environment, requirements, and needs of the customer is vital when designing software.

The Bottom Up approach is as it sounds: you start at the bottom, or in this case the database, and work your way up to the object model. In the LINQ to SQL designer, this is the most supported approach. That isn't to say that you can't use the other approaches with LINQ to SQL (because you can), but the quickest and easiest way is to start with the database and generate your object model from your schema. EF, like LINQ to SQL, supports this approach; however, unlike LINQ to SQL, this is not the primary approach for the EF designer.

Although the DDD people are going to strongly disagree with me, in some situations the Bottom Up approach is as sound as any other development technique. However, it does lend

itself better to situations where you have a normalized, well-thought-out data model, or are designing the entire system from scratch. Nonetheless, in these cases it can be the fastest approach, and if you follow some common design patterns (which are presented in Chapter 2), you can get a pretty good bang for your buck.

Unfortunately, I have to say that it is rare that an organization has an existing well-designed database. My experience over the years puts the estimate of companies with normalized, well-thought-out data models somewhere around 10 percent. The other 90 percent fall somewhere between "Oh boy, they have no referential integrity and are using a single table to store all their data" and "Not too bad—just needs a little normalizing." In these cases, I tend to focus more on the Top Down and the Meet in the Middle approaches.

The Top Down approach, the preferred method of DDD people everywhere, is simply put, modeling your domain on business or conceptual needs rather than the database. This definition is a bit basic and thus is expanded further in later chapters. However, the Top Down approach is the core of DDD. The main drawback to this approach is that it presents a strong learning challenge for people who are not familiar with it and can take some time to implement correctly. However, this approach does allow you to truly model the domain based on specific business needs, thus providing the most flexible design approach.

The Meet in the Middle approach is most applicable to situations in which a database and object model already exist, and your goal is to determine the mappings between the two. This is one of those situations that rarely, if ever, happens because no matter how hard you try, the chances that you have a domain model that orthogonally transects multiple database models is unlikely. Most likely, you will end up refactoring, so this approach really morphs into the Top Down or Bottom Up approach.

In both the Top Down and the Meet in the Middle approaches, LINQ to SQL comes up short. Although the mapping support and the entity support are available, the designer doesn't add much to the equation. Yes, you can drag "classes" from the toolbox, but the functionality is underdeveloped at best. Nevertheless, the designers of LINQ to SQL had the foresight to keep the application programming interface (API) and the internals open enough for us to use the engine with both of these approaches. The Entity Framework designer does a very good job in both of these scenarios; it supports robustly building your conceptual domain model first and supplying the model and the database to build the mappings.

Summary

In this chapter, I have introduced you to some of the basic concepts surrounding ORM, EF, and LINQ to SQL. ORM is the act of connecting object code, whether it is in C#, Java, or any other object-oriented language, to a relational database. This act of mapping is an efficient way to overcome the mismatch that exists between object-oriented development languages and relational databases. Such a mismatch can be classified as an inequality between the native object-oriented language operations and functions and those of a relational database. For example, it is impossible to take an object model and save it directly into a database without some manipulation. This occurs because the database doesn't have the ability to handle inheritance or polymorphism, two basic tenets of object-oriented development. An ORM tool is an excellent solution to overcome the inherent difference between object code and relational databases.

The ORM tool you choose to use in your software should be evaluated based on a set of criteria that meets your goals; however, the following are good starting points: object-to-database mapping, object caching, GUI mapping, portability (multiple DB support), dynamic querying,

lazy loading, nonintrusive persistence, code generation, and stored procedure support. Additionally, along with the criteria and the process direction you use to choose your ORM tool, you should also think about the approach that you envision yourself using. There are three key approaches that you may find useful in your development: the Top Down, Meet in the Middle, and the Bottom Up approaches. Knowing your requirements and your business needs (and wants) will help you choose an appropriate approach and long-term ORM solution.

Chapter 2 more extensively explores the patterns and practices used to create a modular and scalable data access layer using ORM. Chapter 2 does not reinvent these architectural patterns but instead draws upon and simplifies the plethora of existing patterns out there and consolidates them into a single grouping of ORM categories.