Unedited Draft

LINQ IN ACTION

Fabrice Marguerie
Steve Eichert
Jim Wooley

MANNING

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=302

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=302

Contents

Part I - Getting started

1. Introducing LINQ
2. C# and VB.NET language enhancements
3. LINQ building blocks -

Part II - Querying objects in memory

4. Getting familiar with LINQ to Objects
5. Working with LINQ and DataSets
6. Beyond basic in-memory queries

Part III - Manipulating XML

7. Introducing LINQ to XML
8. Querying and transforming XML
9. Common LINQ to XML scenarios

Part IV - Mapping objects to relational databases

10. Getting started with LINQ to SQL
11. Retrieving objects efficiently
12. Advanced LINQ to SQL features

Part V - LINQing it all together

13. Extending LINQ
14. LINQ in every layer

Appendices

Appendix A. The standard query operators

Appendix B. Quick references for VB 8.0 and C# 2.0 features

Features

Appendix C. References

Appendix D. Resources

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=302

Introducing LINQ

Software is simple. It boils down to two things: code and data. Writing software is not so simple, and one of the major activities it involves is programming code to deal with data.

To write code, we can choose from a variety of programming languages. The selected language for an application may depend on the business context, on developer preferences, on the development team's skills, on the operating system or on the company's policy. Whatever the language you end up with, at some point you will have to deal with data. This data can be in files on the disk, tables in a database, XML documents coming from the Web, and very often you have to deal with a combination of all of these. Ultimately, managing data is a requirement for every software project you'll work on.

Given that dealing with data is such a common task for developers, one would expect rich software development platforms like the .NET Framework to provide easy means for this. .NET does provide wide support for working with data. You will see however that something had yet to be achieved: deeper language and data integration. This is where LINQ to Objects, LINQ to XML and LINQ to SQL fit in.

The technologies we present in this book target developers and have been designed as a new way to write code. This book has been written by developers for developers, so don't be afraid, you won't have to wait too long before you are able to write your first lines of LINQ code! In this chapter we will quickly introduce "hello world" pieces of code to give you hints on what you will discover in the rest of the book. The aim is that, by the end of the book, you'll be able to tackle real-world projects while being convinced that LINQ is a joy to work with!

The intent of this first chapter is to give you an overview of LINQ, and help you identify the reasons to use them. We will start by providing an overview of LINQ and the LINQ toolset, which includes LINQ to Objects, LINQ to XML, and LINQ to SQL. We will then review some background information to clearly understand why we need LINQ and where it comes from. The second half of this chapter will guide you while you make your first steps with LINQ code.

1.1 What is LINQ?

Suppose you are writing an application using .NET. Chances are high that at some point you'll need to persist objects to a database, query the database and load the results back into objects. The problem is that in most cases, at least with relational databases, there is a gap between your programming language and the database. Good attempts have been made to provide object-oriented databases, which would be closer to object-oriented platforms and imperative programming languages like C# and VB.NET. However, after all these years, relational databases are still pervasive and you still have to struggle with data-access and persistence in all of your programs.

The original motivation behind LINQ was to address the *impedance mismatch* between programming languages and databases. With LINQ, Microsoft's intention was to provide a solution for the problem of object-relational mapping, as well as simplify the interaction between objects and data sources. LINQ

eventually evolved into a general-purpose language-integrated querying toolset. This toolset can be used to access data coming from in-memory objects (LINQ to Objects), databases (LINQ to SQL), XML documents (LINQ to XML), a file-system, or from any other source.

We will first give you an overview of what LINQ is, before looking at the tools it offers. We will also introduce how LINQ extends programming languages.

1.1.1 Overview

LINQ could be considered as the missing link – whether this pun is intended is yet to be discovered – between the data world and the general-purpose programming languages. LINQ unifies data access, whatever the source of data, and allows mixing data from different kind of sources. LINQ means "Language-INtegrated Query". It allows for query and set operations, similar to what SQL statements offer for databases. LINQ, though, integrates queries directly within .NET languages like C# and Visual Basic through a set of extensions to these languages.

Before LINQ, we had to juggle with different languages like SQL, XML or XPath and various technologies and APIs like ADO.NET or System.Xml in every application written using general-purpose languages like C# or VB.NET. It goes without saying that this had several drawbacks[1]. LINQ kind of glues several worlds together. It helps us avoid the bumps we would usually find on the road from one world to another: using XML with objects, mixing relational data with XML, are some of the tasks that LINQ will simplify.

One of the key aspects of LINQ is that it was designed to be used against any type of objects or data source, and provide a consistent programming model for doing this. The syntax and concepts are the same across all of its uses: once you learn how to use LINQ against an array or a collection, you also know most of the concepts needed to take advantage of LINQ with a database or an XML file.

Another important aspect of LINQ is that when you use it, you work in a strongly-typed world. The benefits include compile-time checking for your queries as well as nice hints from Visual Studio's IntelliSense feature.

LINQ will significantly change some aspects of how you handle and manipulate data with your applications and components. You will discover how LINQ is a step toward a more declarative programming model. Maybe you will wonder in a not so distant future why you had to write so many lines of code…

There is duality in LINQ. You can conceive LINQ as two complementary things: a set of tools that work with data, and a set of programming language extensions.

We will first see how LINQ is a toolset that can be used to work with objects, XML, relational database or other kinds of data. We will then see how LINQ is an extension to programming languages like C# and VB.NET.

1.1.2 LINQ as a toolset

LINQ offers numerous possibilities. It will significantly change some aspects of how you handle and manipulate data with your applications and components. In this book, we'll detail the use of three major flavors of LINQ or *LINQ providers*: LINQ to Objects, LINQ to SQL, and LINQ to XML, respectively in

[1] "It was like you had to order your dinner in one language and drinks in another," said Jason McConnell, Product Manager for Visual Studio at Microsoft. "The direct benefit is programmers are more productive because they have this unified approach to querying and updating data from within their language."

parts 2, 3 and 4 of the book. These three LINQ providers form a family of tools that can be used separately for particular needs or combined together for powerful solutions mixing objects, XML, and relational data.

We will focus on LINQ to Objects, LINQ to SQL and LINQ to XML in this book, but LINQ is open to new data sources. LINQ is not for databases and XML only! The three main LINQ providers listed previously are built on top of a common LINQ foundation. This foundation consists of a set of building blocks like *query operators*, *query expressions* or *expression trees*, which allow the LINQ toolset to be extensible.

Other variants of LINQ can be created to provide access to diverse kinds of data sources. Implementations of LINQ will be released by software vendors. You can also create your own implementations as you'll see in chapter 13, which covers LINQ's extensibility. You can plug anything you like in LINQ to get access to various data sources. This could include the file system, Active Directory, WMI, Windows' Event Log or any other data source or API. This is excellent because it will allow you to benefit from LINQ's features with a lot of the data sources you deal with every day. In fact, Microsoft already offers more LINQ providers that just LINQ to Objects, LINQ to SQL and LINQ to XML. Two of them are LINQ to DataSet, and LINQ to Entities (to work with the ADO.NET Entity Framework). We will present these tools in the second and third parts of this book. For now, let's keep the focus on the big picture.

Here is how we could represent the LINQ building blocks and toolset in a diagram:

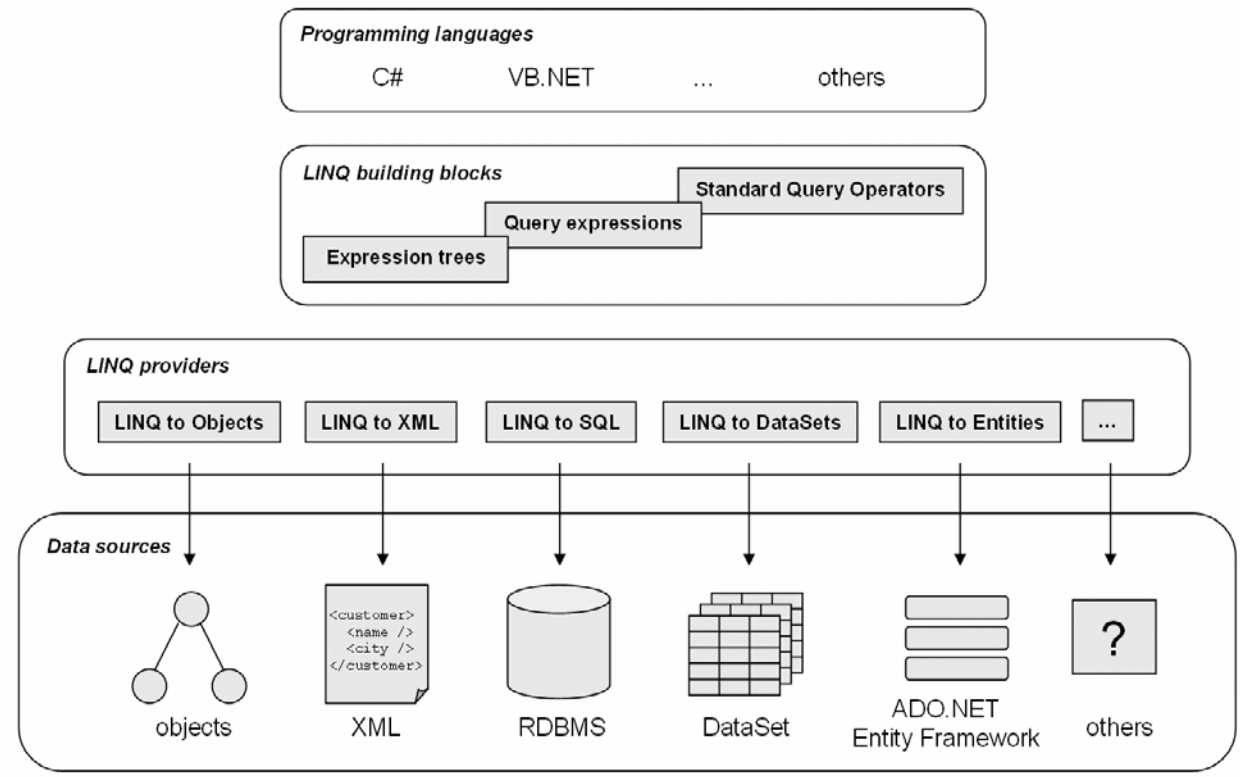

Figure 1.1 LINQ building blocks, LINQ providers and data sources that can be queried using LINQ

The LINQ providers presented in the above diagram are not standalone tools. They are provided as extensions to programming languages. This is the second aspect of LINQ, which is detailed below.

1.1.3 LINQ as language extensions

LINQ allows you to access databases, XML documents and many other data sources by writing queries against these data sources. Rather than being simply *syntactic sugar*[2] that would allow you to easily include SQL queries right into your C# code, LINQ provides you with the same expressive capabilities SQL offers but for your programming language. This is great because a declarative approach like the one LINQ offers allows writing code that is shorter and to the point.

Here is for instance sample C# code you can write with LINQ:

Listing 1.1 Sample code that uses LINQ to query a database and create and XML document

```
// Retrieve customers from a database
var contacts =
  from customer in db.Customers
  where customer.Name.StartsWith("A") && customer.Orders.Count > 0
  orderby customer.Name
  select new { customer.Name, customer.Phone };

// Generate XML data from the list of customers
var xml =
  new XElement("contacts",
    from contact in contacts
    select new XElement("contact",
      new XAttribute("name", contact.Name),
      new XAttribute("phone", contact.Phone)
    )
  );
```

The above piece of code demonstrates all you need to write to extract data from a database and create an XML document from it. Imagine for a moment how you would do the same without LINQ, and you'll realize how things are easier and natural with LINQ. You will soon see more LINQ queries, but let's keep focused on the language aspects for the moment. With the *from*, *where*, *orderby* and *select* keywords all over in the above code, it's obvious that C# has been extended to enable language-integrated queries!

We've just showed you code in C#, but LINQ provides a common querying architecture across programming languages. It works with C# 3.0 and VB.NET 9.0, and as such requires dedicated compilers, but it can be ported to other .NET languages. This is already the case for F#, a functional language for .NET from Microsoft Research. It will also be the case for the Borland Delphi language, for example, and more languages are expected to support LINQ.

The following diagram shows a typical language-integrated query that is used to talk to objects, XML or data tables:

[2] *Syntactic sugar* is a term coined by Peter J. Landin for additions to the syntax of a computer language that do not affect its expressiveness but make it "sweeter" for humans to use. Syntactic sugar gives the programmer an alternative way of coding that is more practical, either by being more succinct or more like some familiar notation.

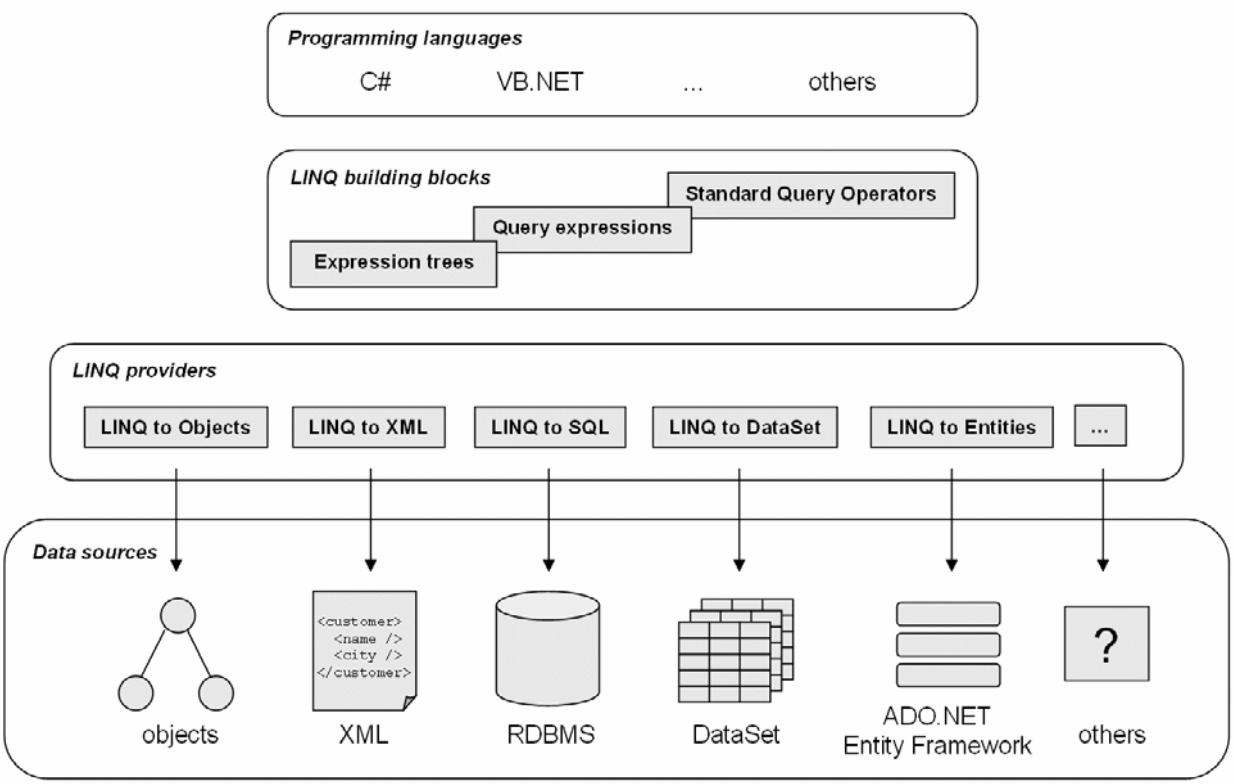

Figure 1.2 LINQ as language extensions and as a gateway to several data sources

The query in the above diagram is expressed in C#, and not in a new language. LINQ is not a new language. It is integrated in C# and VB.NET. In addition, LINQ can be used to avoid entangling your .NET programming language with SQL, XSL, or other data-specific languages. It's the set of language extensions coming with LINQ that enables queries over several kinds of data stores to be formulated right into programming languages. You can think of LINQ as a universal remote control, if you wish. At times, you'll use it to query a database; at others, you'll query an XML document; etc. But you'll do all this in your favorite language, without having to switch to another one like SQL or XSLT.

In chapter 2, we'll show you the details of how the programming languages have been extended to support LINQ. In chapter 3, you'll learn how to write LINQ queries. This is where you'll learn about *query operators, query expressions*, and *expression trees*. But, we still have a few things to discover before getting there…

Now that we have given you and idea of what LINQ is, let's discuss the motivation behind it, and then we'll review its design goals and a bit of history.

1.2 Why do we need LINQ?

We have just provided you with an overview of LINQ. The big question at this point is: why would we like, in the first place, to have a tool that makes working with programming languages, relational data and XML at the same time more convenient?

At the origin of the LINQ project is a simple fact: How many applications access data or talk to a SQL database? The answer: the vast majority of them! Most applications deal with relational databases.

Consequently, in order to program applications, learning a language like C# is not enough. You also have to learn SQL and the APIs that tie together C# and SQL to form your full application.

We'll start by taking a look at a piece of data-access code that uses the standard .NET APIs. This will allow us to point out the common problems that are encountered in this kind of code. We will then extend our analysis to a higher level by showing how these problems exist with other kinds of data such as XML. You'll see that LINQ addresses a general *impedance mismatch* between data sources and programming languages. Finally a piece of short code sample will give you a glimpse at how LINQ is a solution to the problem.

1.2.1 Common problems

The recurrence of database connectivity in applications requires that the .NET Framework address the need for developers to write code to access data. Of course this is the case since the first appearance of .NET. The .NET Framework Class Library (the FCL) includes ADO.NET. ADO.NET provides an API to get access to relational databases and to represent relational data in memory. This API consists of classes like *SqlConnection*, *SqlCommand*, *SqlReader*, *DataSet* and *DataTable*, just to name a few. The problem with these classes is that they force the developer to work explicitly with tables, records and columns, while modern languages like C# and VB.NET use object-oriented paradigms. We are going to see below an example of this with some code samples. By looking at the problems that exist with traditional code, you'll be able to see how LINQ comes to the rescue.

In fact, now that the object-oriented paradigm is adopted as the prevailing model in software development, developers incur a large amount of overhead in mapping it to other abstractions, specifically relational databases and XML. The result is that a lot of time is spent on writing plumbing code[3]. Removing this burden would increase the productivity in data-intensive programming, which LINQ helps us do.

But wait, it's not only about productivity! It also impacts quality. Writing tedious and fragile code like plumbing code can lead to insidious defects in software or degraded performance.

Let's take a look at a short piece of code that shows how we would typically access a database in a .NET program:

Listing 1.2 Typical .NET data-access code

```
using (SqlConnection connection = new SqlConnection("..."))
{
  connection.Open();
  SqlCommand command = connection.CreateCommand();
  command.CommandText =                                          |#1
    @"SELECT Name, Country                                       |#1
      FROM Customers                                             |#1
      WHERE City = @City";                                       |#1
  command.Parameters.AddWithValue("@City", "Paris");             |#2
  using (SqlDataReader reader = command.ExecuteReader())
  {
    while (reader.Read())
    {
      string name = reader.GetString(0);                         |#3
```

[3] It is estimated that dealing with the task of storing and retrieving objects to and from data stores accounts for between 30 and 40 percent of a development team's time.

```
      string country = reader.GetString(1);                    |#3
      ...
    }
  }
}
```

(Annotation) <#1 SQL query in a string
(Annotation) <#2 Loosely-bound parameters
(Annotation) <#3 Loosely-typed columns

Just by taking a quick look at this code, we can list several limitations of the model:

While we want to perform a simple task, several steps and verbose code are required

Queries are expressed as quoted strings [#1], which means they bypass all kinds of compile-time checks. What if the string does not contain a valid SQL query? What if a column has been renamed in the database?

The same applies for the parameters [#2] and for the result sets [#3]: they are loosely-defined. Are the columns of the type we expect? Also, are we sure we use the correct number of parameters? Are the names of the parameters in sync between the query and the parameter declarations?

The classes we use are dedicated to SQL Server and cannot be used with another database server.

Of course, other solutions already exist. We could use a code generator or one of the several object-relational mapping tools available around. The problem is that these tools are not perfect either, and they have their own limitations. For example, if they are designed for accessing databases, most of the time they don't deal with other data sources like XML documents. Also, one thing that Microsoft can do that other vendors can't is integrate data access and querying features right into the C# and VB.NET languages.

The motivation for LINQ is two-fold: Microsoft did not have a data-mapping solution yet, and with LINQ it had the opportunity to integrate the mapping and querying into the programming languages. This could remove most to the limitations we identified in Listing 1.2.

The main idea is that by using LINQ you are able to gain access to any source of data by writing queries, like the following, directly in the programming language that you master and use every day:

Listing 1.3 Simple query expression

```
from customer in customers
where customer.Name.StartsWith("A") && customer.Orders.Count > 0
orderby customer.Name
select new { customer.Name, customer.Orders }
```

In this query, the data could be in memory, in a database, in an XML document or in another place; the syntax would remain similar. As we saw in Figure 1.2, this kind of query can be used with multiple types of data and different data sources, thanks to LINQ's extensibility features. For example, in the future we are likely to see appear an implementation of LINQ to program queries against a file system or to call web services.

1.2.2 Addressing a paradigm mismatch

Let's continue looking at why we need LINQ. The fact that in modern applications we have to deal at the same time with general-purpose programming languages, relational data, SQL, XML documents, XPath and so on means that we need two things:

1. to be able to work with any of these technologies or languages individually,

2. to mix and match them to build a rich and coherent solution.

The problem is that object-oriented programming, the relational database model and XML – just to name a few – were not originally built to work together. They represent different paradigms that don't play well one with another.

What is this impedance mismatch everybody's talking about?

Data is generally manipulated by application software written using object-oriented programming languages such as C#, VB.NET, Java, Delphi or C++. But translating an object graph into another representation, such as tuples of a relational database, often requires tedious code.

The general problem LINQ addresses could be stated like this: *"Data != Objects"*. More specifically, for LINQ to SQL: *"Relational data != Objects"*. The same could apply for LINQ to XML: *"XML data != Objects"*. We should also add: *"XML data != Relational data"*.

You've probably heard the term *impedance mismatch* before. It is a term that's commonly applied to the incompatibility between systems. *Impedance mismatch* describes an inadequate ability of one system to accommodate input from another. Although the term originated in the field of electrical engineering, it has been generalized and used as a term of art in systems analysis, electronics, physics, computer science and informatics.

Object-relational mapping

If we take the object-oriented paradigm and the relational paradigm, the mismatch exists at several levels. Let's just name a few.

Relational databases and object-oriented languages don't share the same set of primitive data types. For example, strings usually have a delimited length in databases, which is not the case in C# or VB.NET. This can be a problem if you try to persist a 150-character string in a table field that accepts only 100 characters. Another simple example is that most databases don't have a Boolean type, while we frequently use true/false values in programming languages.

OOP and relational theories come with different data models. For performance reasons and due to their intrinsic nature, relational databases need to be normalized. Normalization is a process that eliminates redundancy, organizes data efficiently, and reduces the potential for anomalies during data operations and improves data consistency. Normalization results in an organization of data that is specific to the relational data model. This prevents a direct mapping of tables and records to objects and collections. Relational databases are normalized in tables and relations, while objects use inheritance, composition and complex reference graphs. A common problem exists because relational databases don't have concepts like inheritance: mapping a class hierarchy to a relational database requires using "tricks".

Programming models. In SQL you write queries, and so you have a higher-level declarative way of expressing the set of data that you're interested in. With general purpose imperative programming languages like C# or VB.NET, you've got to write for loops and if statements and so forth.

Encapsulation. Objects are self-contained and include data as well as behavior. In databases, data records don't have behavior per se. It's possible to act on database records only through the use of SQL queries or stored procedures. In relational databases, code and data are clearly separated.

The mismatch is a result of the differences between a normalized relational database and a typical object oriented class hierarchy. One might say relational databases are from Mars and objects are from Venus.

Let's take the simple example shown in Figure 1.3. We have an object model we'd like to map to a relational model:

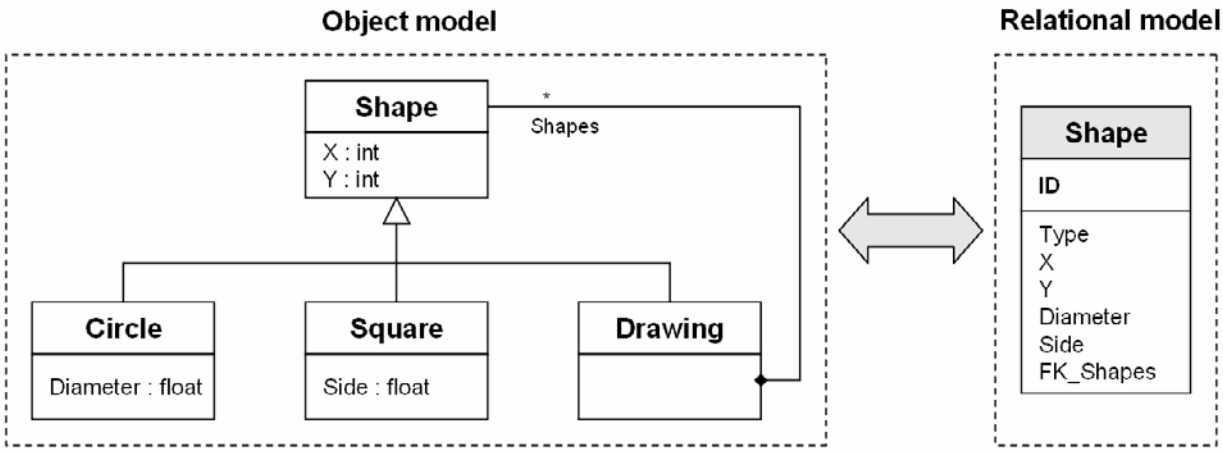

Figure 1.3 How simple objects can be mapped to a database model. The mapping is not trivial due to the differences between the object-oriented and the relational paradigms.

Concepts like inheritance or composition are not directly supported by relational databases, which means that we cannot represent the data in the same way in both models. You can see here that several objects and types of objects can be mapped to a single table.

Even if we wanted to persist an object model like the one we have here in a new relational database, we would not be able to use a direct mapping. For instance, for performance reasons and to avoid duplication, it's much better in the present case to create only one table in the database. A consequence of doing so, however, is that data coming from the database table cannot be used without effort to repopulate an object graph in memory. As you can see, when you win on one side, you lose on the other.

We may be able to design a database schema or an object model to reduce the mismatch between both worlds, but we'll never be able to remove it because of the intrinsic differences between the two paradigms. We don't even always have the choice. Quite often, the database schema is already defined in advance, and in some cases we have to work with objects defined by someone else.

The complex problem of data source integration with programs involves more than simply reading from and writing to a data source. When programming using an object-oriented language, we usually want our applications to use an object model that is a conceptual representation of the business domain, instead of being tied directly to the relational structure. The problem is that at some point we need to make the object model and the relational model work together. This is not easy at all because object-oriented programming languages and .NET involve entity classes, business rules, complex relationships, and inheritance, while a relational data source involves tables, rows, columns, and primary and foreign keys…

A typical solution for bridging object-oriented languages and relational databases is *object-relational mapping*. This refers to the process of mapping your relational data model to your object model, usually back and forth. Mapping can be defined as: the act of determining how objects and their relationships are persisted in permanent data storage, in this case relational databases.

Databases[4] do not map naturally to object models. Object-relational mappers are automated solutions to address the impedance mismatch. To make a long story short: you provide an object-relational mapper with your classes, your database, and the mapping configuration, and it takes care of the rest. It generates the SQL queries, fills your objects with data from the database, persists them in the database, etc.

As you can guess, no solution is perfect and object-relational mappers could be improved. Some of their main limitations include:

- a good knowledge of the tools is required before being able to use them efficiently and avoid performance issues,
- an optimal use still requires knowledge on how to work with a relational database,
- mapping tools are not always as efficient as hand-written data-access code,
- not all the tools come with support for compile-time validation.

Multiple object-relational mapping tools are available for .NET. There is a choice of Open Source, free or commercial products. As an example, here is a mapping configuration file for NHibernate, which is one of the Open Source mappers:

```
<?xml version="1.0" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" namespace="Eg" assembly="Eg">
  <class name="Cat" table="CATS" discriminator-value="C">
    <id name="Id" column="uid" type="Int64">
      <generator class="hilo"/>
    </id>
    <discriminator column="subclass" type="Char"/>
    <property name="Birthdate" type="Date"/>
    <property name="Color" not-null="true"/>
    <property name="Sex" not-null="true" update="false"/>
    <property name="Weight"/>
    <many-to-one name="Mate" column="mate_id"/>
    <set name="Kittens">
      <key column="mother_id"/>
      <one-to-many class="Cat"/>
    </set>
    <subclass name="DomesticCat" discriminator-value="D">
      <property name="Name" type="String"/>
    </subclass>
  </class>

  <class name="Dog">
    <!-- mapping for Dog could go here -->
  </class>
</hibernate-mapping>
```

Figure 1.4 NHibernate mapping file that is used to map a Cat class to a CATS table in a relational database. Fields, relationships, and inheritance are defined using XML.

[4] We are talking only about relational databases here because this is what is used in the vast majority of business application throughout the world. Object-oriented databases offer a different approach that allows persisting objects more easily. Whether object-oriented databases are better than relational databases is another debate, which we are not going to address in this book.

In part 3 of this book, you'll see how LINQ to SQL is an object-relational mapping solution and how it addresses some of the issues listed above. But for now, we are going to look at another problem LINQ can solve.

Object-XML mapping

Analogous to the object-relational impedance mismatch, a similar mismatch also exists between objects and XML. For example, the type system part of the W3C XML Schema specification has no one-to-one matching with the type system of the .NET Framework for example. Using XML in a .NET application is not so much of a problem because we already have APIs that deal with this under the *System.Xml* namespace and built-in support for object to/from XML serialization and deserialization. Still, a lot of tedious code is required most of the time for doing even simple things on XML documents.

Given that XML has become so pervasive in the modern software world, something had to be done to reduce the work required to deal with XML in programming languages.

When you look at these domains, it is remarkable how different they are. The main source of contention relates to the fact that:

- Relational databases are based on relation algebra and are all about tables, rows, columns, SQL, queries, etc.
- XML is all about text, angle brackets, elements, attributes, hierarchical structures, etc.
- Object-oriented general-purpose programming languages and the .NET Framework CLR live in a world of classes, methods, properties, inheritance, etc.

Many concepts are specific to each domain and have no direct mapping to another domain. The following picture gives an overview of the concepts used in .NET and object-oriented programming, in comparison to the concepts used in data sources such as XML documents or relational databases:

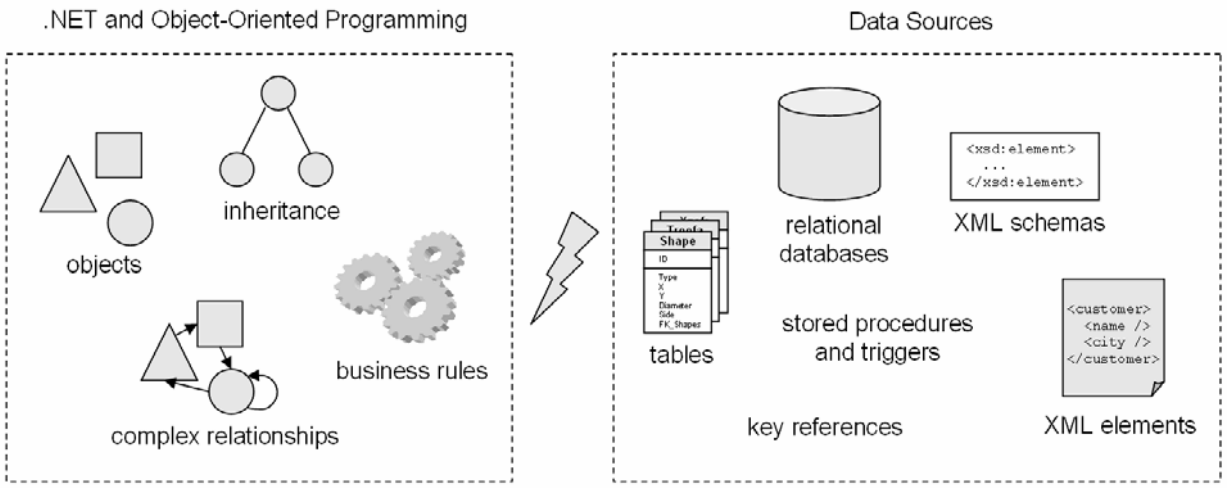

Figure 1.5 .NET applications and data sources are different worlds. The concepts used in object-oriented programming are different from the concepts used with relational databases and XML.

Too often, programmers have to do a lot of plumbing work to tie together the different domains. Different APIs for each data type cause developers to spend an inordinate amount of time to learn, write, debug, and rewrite brittle code. The usual culprits that break the pipes are bad SQL query strings or XML tags or content that doesn't get checked until run time. .NET languages like C# and VB.NET assist the

developers a lot and provide such things as IntelliSense, strongly-typed code, compile-time checks, still this can get broken if you start to include malformed SQL queries or XML fragments in your code, none of which are validated by the compiler.

A successful solution requires bridging the different technologies, and solving the object-persistence impedance mismatch – a challenging and resource-intensive problem. To solve this problem, we must resolve the following issues between .NET and data source elements:

- Fundamentally different technologies
- Different skill sets
- Different staff and ownership for each of the technologies
- Different modelling and design principles

Some efforts have been made to reduce the impedance mismatch by bringing some pieces of one world into another one. For example: SQLXML 4.0 ties SQL to XSD; System.Xml spans XML/ XML DOM/XSL/XPath and CLR; the ADO.NET API bridges SQL and CLR data types; SQL Server 2005 includes CLR integration. All these efforts are proofs that data integration is essential, however they represent distinct moves without a common foundation, which makes them difficult to use together. LINQ, in contrast, offers a common infrastructure to address the impedance mismatches.

1.2.3 LINQ to the rescue

To succeed in using objects and relational databases together you need to understand both paradigms, and their differences, and then make intelligent tradeoffs based on that knowledge. The main goal of LINQ and LINQ to SQL is to get rid of, or at least reduce, the need to worry about these limits.

An impedance mismatch forces you to choose one side or the other as the "primary" side. With LINQ, Microsoft chooses the programming language side, because it's easier to adapt the C# and VB.NET languages than to change SQL or XML. With LINQ, what Microsoft is really aiming at is deeply integrating the capabilities of data query and manipulation languages into programming languages.

LINQ removes many of the barriers between objects, databases and XML. It enables us to work with each of these paradigms using the same language-integrated facilities. For example, we are able to work with XML data and data coming from a relational database within the same query.

Since code is worth a thousand words, let's take a look at a quick code sample using the power of LINQ to retrieve data from a database and create an XML document in a single query. The following piece of code creates an RSS feed based on relational data:

Listing 1.4 Working with relational data and XML in the same query

```
var database = new RssDB("server=localhost; initial catalog=RssDB");

XElement rss = new XElement("rss",
  new XAttribute("version", "2.0"),
  new XElement("channel",
    new XElement("title", "LINQ in Action RSS Feed"),
    new XElement("link", "http://LinqInAction.net"),
    new XElement("description", "The RSS feed for this book"),
    from post in database.Posts
    orderby post.CreationDate descending
    select new XElement("item",
      new XElement("title", post.Title),
```

```
      new XElement("link", "posts.aspx?id="+post.ID),
      new XElement("description", post.Description),
      from category in post.Categories
      select new XElement("category", category.Description)
    )
  )
);
```

We won't detail here how this code works. You'll see documented examples like this one in parts 3 and 4 of the book. What is important to note at this point is how LINQ makes it easy to work with relational data and XML in the same piece of code. If you have already done this kind of work before, it should be obvious to you that this code is very concise and readable in comparison to the solutions at your disposal before LINQ appeared.

Before seeing more code samples and helping you write your own LINQ code, we'll quickly review where LINQ comes from. It's interesting to know where LINQ takes his roots from, and understand the links with other projects from Microsoft you may have heard of. It's also important to know clearly what Microsoft set to achieve with LINQ. This is why we'll start the next section by reviewing the design goals of the LINQ project, and we'll then spend some time on its history.

1.3 Design goals and origins of LINQ

LINQ was designed to address the mismatch between the different kinds of data. In order to better understand what services it provides, we will sum up the design goals of the LINQ project.

We will also take a look at the history of the LINQ project to know how it was born. LINQ is not a recent project from Microsoft in the sense that it inherits a lot of features from research and development work done by Microsoft over the last years. We'll see the relationships LINQ has with other Microsoft projects so you know if LINQ replaces projects like C , ObjectSpaces, WinFS or support for XQuery in the .NET Framework.

1.3.1 The goals of the LINQ project

Let's review the design goals Microsoft set for the LINQ project in order to get a clear understanding of what LINQ offers:

Table 1.1 LINQ's design goals and the motivations behind them

Goal	Motivation
Integrate objects, relational data, and XML	Unified query syntax across data sources to avoid different languages for different data sources. Single model for crunching all types of data regardless of source or in-memory representation.
SQL and XQuery-like power in C# and VB	Integrate querying abilities right into the programming languages.
Extensibility model for languages	Enable implementation for other programming languages.
Extensibility model for multiple data sources	Be able to access other data sources than relational databases or XML documents. Allow other frameworks to implement LINQ for their own needs.
Type safety	Compile-time type checking to avoid problems that were previously discovered at run-time only.

	The compiler will catch errors in your queries.
Extensive IntelliSense support (enabled by strong-typing)	Assist the developers when writing queries to improve productivity and to help them get up to speed with the new syntax. The editor will guide you when writing queries.
Debugger support	Allow the developers to debug LINQ queries step by step and with rich debugging information.
Build on the foundations laid in C# 1.0 and 2.0, VB.NET 7.0 and 8.0	Reuse the rich features that have been implemented in the previous versions of the languages.
Run on the .NET 2.0 CLR	Avoid requiring a new runtime and creating unnecessary deployment hassles.
Remain 100% backwards compatible	Be able to use standard and generic collections, data-binding, existing Windows Forms and web controls, etc.

The number one LINQ feature is the ability to deal with several data types and sources. LINQ ships with implementations that support querying against regular object collections, databases, entities, and XML sources. Because LINQ supports rich extensibility, developers can also easily integrate it with other data sources and providers.

Another essential feature of LINQ is that it is strongly-typed. This means that:

- You get compile-time checking for all queries. Unlike SQL statements today, where you typically only find out at run-time if something is wrong, this means you will be able to check during development that your code is correct. The direct benefit is a reduction of the number of problems discovered late in production. Most of the time, issues come from human factors. Strongly-typed queries allow detecting early typos and mistakes done by the developer in charge of the keyboard…
- You will get IntelliSense within Visual Studio when writing LINQ queries. This makes both typing faster, but also make it much easier to work against both simple and complex collection and data source object models.

This is all well and sounds great, but where does LINQ come from? Before delving into LINQ and starting to use it, let's see how it was born.

1.3.2 A bit of history

LINQ is the result of a long-term research process inside Microsoft. Several projects involving evolutions of programming languages and data-access methods can be considered as the parents of LINQ to Objects, LINQ to XML (formerly known as XLinq) and LINQ to SQL (formerly known as DLinq).

LINQ builds heavily on research projects carried out by Microsoft Research in Cambridge and Redmond. The main project is named C (or the C-Omega language).

C-Omega

C is an extension of the C# language in two areas:

A control flow extension for asynchronous wide-area concurrency (formerly known as Polyphonic C#),

A data type extension for XML and table manipulation (formerly known as Xen and as X#).

C covered more than what comes with LINQ, but a good deal of what is now included as part of the LINQ technologies was already present in C . The C project was conceived as experimenting with integrated

queries, sort of mixing C# and SQL, C# and XQuery, etc. This part was covered by researchers like Erik Meijer, Wolfram Schulte and Gavin Bierman, who published multiple papers on the subject.

C was released as a preview in 2004. A lot has been learned from that prototype, and a few months later, Anders Hejlsberg, chief designer of the C# language announced that Microsoft would be working on applying a lot of that knowledge in C# and other programming languages. Anders said at that time that his particular interest for the past couple of years had been to think deeply about the big impedance mismatch between programming languages, C# in particular, and the data world. This includes database and SQL, but also XML and XQuery, for example.

C extensions to the .NET type system and to the C# language were the first steps to a unified system that treats SQL-style queries, query result sets, and XML content as full-fledged members of the .NET Framework. C introduced the stream type, which is analogous to the .NET Framework 2.0's *System.Collections.Generic.IEnumerable<T>* type. C also defined constructors for typed tuples (called anonymous structs), which are similar to the anonymous types we get in C# 3.0 and VB.NET 9.0. Another thing C supported is embedded XML, something we are able to see in VB.NET 9.0.

Another project with a strong relationship with LINQ is ObjectSpaces, an attempt Microsoft made in the field of object-relational mapping.

ObjectSpaces

LINQ to SQL is not Microsoft's first attempt at object-relational mapping.

The first preview of a project named ObjectSpaces appeared in a PDC 2001 ADO.NET presentation. ObjectSpaces was a set of data access APIs. It allowed data to be treated as objects, independent of the underlying data store. ObjectSpaces also introduced OPath, a proprietary object query language. In 2004, Microsoft announced that ObjectSpaces was depending on the WinFS[5] project, and as such would be postponed to the Orcas timeframe (the next releases after .NET 2.0 and Visual Studio 2005). No new releases happened after that. Everybody realized that ObjectSpaces would never see the light of day when Microsoft announced that WinFS wouldn't make it to the first release of Windows Vista.

XQuery implementation

Similarly to what happened with ObjectSpaces and at about the same time, Microsoft had started working on an XQuery processor. A preview was included in the first beta release of the .NET Framework version 2.0, but Microsoft finally decided not to ship a client-side[6] XQuery implementation in the final version of .NET Framework 2.0. One problem with XQuery is that it is an additional language you have to learn specifically to deal with XML…

Why all these steps back? Why did Microsoft apparently stop working on these technologies? Well, the cat came out of the bag at the PDC 2005, when the LINQ project was announced.

LINQ has been designed by Anders Hejlsberg to address this impedance mismatch right into the programming languages like C# and VB.NET. With LINQ, you can query pretty much anything. LINQ works with any collection you have in the .NET Framework. It's a single programming model for data

[5] WinFS was a project for a relational file system Microsoft had been developing for Windows. It has been cancelled in 2006.

[6] A server-side implementation of XQuery is included inside SQL Server 2005 though. Now that the XQuery standard has been finalized, Microsoft is considering again whether to add support for XQuery in .NET.

wherever this data comes from. This is why Microsoft favored LINQ instead of continuing investing in separate projects like C , ObjectSpaces or the support for XQuery.

As we have just seen, LINQ has a rich history behind it and has benefited from all the research and development work done on the now defunct projects. Before going further and discovering how it works, how to use it, and its different flavors, what about taking a look at some code?

The next three sections of this chapter provide simple code that demonstrates LINQ to Objects, LINQ to XML and LINQ to SQL. This will give you an overview of what LINQ code looks like and show you how it can help you work with object collections, XML and relational data.

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

After this introduction, you are probably eager to look at some code and to make your first steps with LINQ. We think that you'll get a better understanding of the features LINQ provides if you can spend some time on a piece of code. This is what this book is about, anyway!

We'll first check that you have everything you need to be able to use LINQ code, and then we'll introduce our first detailed LINQ code sample.

1.4.1 What you need to get started

Before looking at code, let's spend some time reviewing all you need to be able to test this code.

Compiler and .NET Framework support

LINQ features are a matter of compiler and libraries, not runtime. It is important to understand that while the C# and VB.NET languages have been enriched and a few new libraries have been added to the .NET Framework, the .NET Runtime (the CLR) did not need to evolve. New compilers are required for C# 3.0 and VB.NET 9.0, but the required runtime is still an unmodified version 2.0.

Required software

LINQ is delivered as an integral part of the.NET Framework version 3.5. This version of the framework comes with additional and updated libraries, as well as new compilers for the C# and VB.NET languages, but it stays 100% compatible with the .NET Framework 2.0. This means that the applications you'll build using LINQ can run in a "bare" .NET 2.0 runtime!

Add a word about the support from Silverlight and the DLR

To be updated when Microsoft announces a clear schedule

LINQ is delivered with .NET as part of the *Orcas*[7] wave. This will include new versions of Visual Studio, the .NET Framework, C# and VB.NET. The official release of Orcas is due for 2007.

- Time frames and availability
  - LINQ preview releases (Beta 2)
  - The roadmap to LINQ

To setup your machine and be able to run our code samples as you read, you only need to install:

[7] *Orcas* is the code name for the next version of Visual Studio and the .NET Framework.

- At least one of these versions of Visual Studio:
  - Visual C# "Orcas" Express Edition
  - Visual Basic "Orcas" Express Edition
  - Visual Web Developer "Orcas" Express Edition
  - Visual Studio "Orcas" Standard Edition or higher
- If you want to run the LINQ to SQL samples, one of the following is required:
  - SQL Server 2005 Express Edition
  - SQL Server 2005
  - SQL Server 2000a
  - or a later version of SQL Server

Language considerations

In this book, we'll suppose you know the syntax of the C# programming language and occasionally of VB.NET. We will go through the syntax evolutions provided by C# 2.0, C# 3.0, VB.NET 9.0 and LINQ in this book, but for the sake of simplicity we'll skip this for our first few code samples and keep that for chapters 2 and 3. We believe that LINQ's query syntax is simple enough for anyone to understand our first code samples.

Note

Most of the examples contained in this book are in C#, but they can easily be ported to VB.NET as the syntax is similar between the two languages.
Code examples will be in VB.NET when we will examine the features specific to this language or simply when it makes sense.

Alright, enough preliminaries, let's look at some code! Let's dive immediately into a simple example that will show you how to query a collection in memory using LINQ to Objects. Follow the guide, and just be receptive to the magic of all these new features you'll use very soon in your own applications!

1.4.2 Hello LINQ to Objects

This may be your first contact with the new concepts and syntactic constructs. No fear, no stress! Our ultimate goal is for you to master these technologies, but don't force yourself to understand everything at once. We'll take the time we need to come back to every single detail of LINQ and the new language extensions as we progress through the book.

Here is our first LINQ example in C#:

Listing 1.5 Hello LINQ in C#

C#

```
using System;
using System.Linq;

static class HelloWorld
{
  static void Main()
  {
    string[] words = { "hello", "wonderful", "linq", "beautiful", "world" };

    // Get only short words
```

```
    var shortWords =
      from word in words
      where word.Length <= 5
      select word;

    // Print each word out
    foreach (var word in shortWords)
      Console.WriteLine(word);
  }
}
```

The same example in VB.NET:

Listing 1.6 Hello LINQ in VB.NET

VB

```
Module HelloWorld
  Sub Main()
    Dim words As String() = { "hello", "wonderful", "linq", "beautiful", "world" }

    ' Get only short words
    Dim shortWords = _
      From word In words _
      Where word.Length <= 5 _
      Select word

    ' Print each word out
    For Each word In shortWords
      Console.WriteLine(word)
    Next
  End Sub
End Module
```

Note

Most of the code examples contained in this book can be copied and pasted without modification into a console application for testing.

If you were to compile and run these codes, here is the output you'd see:

```
hello
linq
world
```

The code speaks for itself. We filter a list of words to select only the ones which length is less than or equal to five characters.

One could argue that the same result could be achieved without LINQ using the following code:

Listing 1.7 Old-school version of Hello LINQ

C#

```
using System;

static class HelloWorld
{
  static void Main()
```

```
  {
    string[] words = new string[] { "hello", "wonderful", "linq",
                                    "beautiful", "world" };

    foreach (string word in words)
    {
      if (word.Length <= 5)
       Console.WriteLine(word);
    }
  }
}
```

Notice how this "old-fashioned" code is much shorter than the LINQ version and very easy to read… Well, don't give up with this book just yet. There is much more to LINQ than what we show in this first simple program!

If you get along with us past this paragraph, we will be able to help you discover all the power of LINQ to Objects, LINQ to SQL and LINQ to XML.

Just to give you a motivation to pursue reading, let's try to improve our simple example with grouping and sorting. This should give you an idea of why LINQ is useful and powerful.

In order to get the following result:

```
Words of length 9
  beautiful
  wonderful
Words of length 5
  hello
  world
Words of length 4
  linq
```

We can use the following C# code:

Listing 1.8 Hello LINQ in C# improved with grouping and sorting

C#

```
using System;
using System.Linq;

static class HelloWorld
{
  static void Main()
  {
    string[] words = { "hello", "wonderful", "linq", "beautiful", "world" };

    // Group words by length
    var groups =
      from word in words
      orderby word ascending
      group word by word.Length into lengthGroups
        orderby lengthGroups.Key descending
        select new { Length = lengthGroups.Key, Words = lengthGroups };
```

```
    // Print each group out
    foreach (var group in groups)
    {
      Console.WriteLine("Words of length "+group.Length);
      foreach (string word in group.Words)
        Console.WriteLine("   "+word);
    }
  }
}
```

Or this equivalent VB.NET code:

Listing 1.9 Hello LINQ in VB.NET improved with grouping and sorting

VB

```
Module HelloWorld
  Sub Main()
    Dim words as String() = {"hello", "wonderful", "linq", "beautiful", "world"}

    ' Group words by length
    Dim groups = _
      From word In _
        (From w In words _
        Select w _
        Order By w) _
      Group By word.Length _
      Select Length = It.Key, Words = It

    ' Print each group out
    For Each Dim g In groups
      Console.WriteLine("Words of length " + g.Length.ToString())
      For Each Dim word In g.Words
        Console.WriteLine("   " + Word)
      Next
    Next
  End Sub
End Module
```

"Group By" is not implemented yet in VB Orcas Beta 1! It will be in Beta 2.

What happens here is that we have expressed in one query (or two nested queries more precisely) what could be formulated in English as: “Sort words from a list alphabetically and group them by their length in descending order”.

We’ll leave doing the same without LINQ as an exercise for the reader… If you take the time to do it, you’ll notice that it takes more code and requires dealing a lot with collections. One first advantage of LINQ that stands out with this example is the power of expressiveness it enables: you can express declaratively what you want to achieve using queries instead of actually writing convoluted pieces of code.

We won’t take time right now to get into the details of the code you’ve just seen. Whoever is familiar with SQL probably already gets a good understanding of this code. In addition to all the nice SQL-like querying, LINQ also provides a number of other “functions” such as *Sum*, *Min*, *Max*, *Average*, and much more. This allows performing a rich set of operations.

Quick example:

```
decimal totalAmount = orders.Sum(order => order.Amount);
```

Here we sum the amount of each order in a list of orders to compute a total amount.

We'll make sure that everyone of you completely understands this kind of code later in this book. However, before we continue, some of you may wish to test our "Hello LINQ" example and start playing with the code. Feel free to do so to get an idea of how easy LINQ really is.

Once you are ready, let's move on to LINQ to XML and LINQ to SQL. We'll spend some time with these two other flavors of LINQ so you can get an idea of what they taste like. We will get back to LINQ to Objects in detail in part 2 of this book.

1.5 First steps with LINQ to XML: Querying XML documents

As we said in the first half of this chapter, the extensibility of the LINQ query architecture is used to provide implementations that work over both XML and SQL data. We will now help you to make your first steps with LINQ to XML.

LINQ to XML is a member of the LINQ family of technologies: it provides XML querying capabilities integrated into host programming languages. It takes advantage of the LINQ framework and adds query extensions specific to XML. LINQ to XML provides the query and transformation power of XQuery and XPath integrated into .NET.

From another perspective you can also think of LINQ to XML as a full-featured XML API comparable to a modernized, redesigned *System.Xml* API plus a few key features from XPath and XSLT. LINQ to XML provides facilities to edit XML documents and element trees in-memory, as well as streaming facilities. This means that you'll be able to use LINQ to XML to perform more easily many of the XML processing tasks that you have been performing with the traditional XML APIs from the *System.Xml* namespace.

We will first examine why we need an XML API like LINQ to XML by comparing it to some alternatives. We will then make our first steps with some code using LINQ to XML in our obligatory "Hello World" example.

1.5.1 Why we need LINQ to XML

XML is ubiquitous nowadays and is used extensively in applications written using general-purpose languages like C# or VB.NET. It is used to exchange data between applications, to store configuration information, to persist temporary data, as a base for generating web pages or reports, and for many other things. It is everywhere!

Until now, XML hasn't been natively supported by the programming languages, which required using APIs to deal with XML data. These APIs include *XmlDocument*, *XmlReader*, *XPathNavigator*, *XslTransform* for XSLT, and SAX and XQuery implementations. The problem is that these APIs are not well integrated with the programming languages, often requiring several lines of unnecessarily-convoluted code to achieve a simple result. You'll see an example of this in the next section (Listing 1.12). But for the moment, let's see what LINQ to XML has to offer.

LINQ to XML extends the language-integrated query features offered by LINQ to add support for XML. It offers the expressive power of XPath and XQuery but with C# or VB as the programming language and with type safety and IntelliSense.

If you worked on XML documents with .NET, you probably used the XML DOM (Document Object Model) available through the *System.Xml* namespace. LINQ to XML leverages the experience with the DOM to improve the developer toolset and avoid the limitations of the DOM.

Let's compare the characteristics of LINQ to XML with the ones of the XML DOM:

Table 1.2 Comparing LINQ to XML with the XML DOM to show how LINQ to XML is a better value proposition

LINQ to XML characteristics	XML DOM characteristics
Element-centric	Document-centric
Declarative model	Imperative model
LINQ to XML code presents a layout close to the hierarchical structure of an XML document	No resemblance between code and document structure
Language-integrated queries	No integrated queries
Creating elements and attributes can be done in one instruction; Text nodes are just strings	Basic things require a lot of code
Simplified XML namespace support	Requires dealing with prefixes and "namespace managers"
Faster and smaller	Heavyweight and memory intensive
Streaming capabilities	Everything is loaded in memory
Symmetry in element and attribute APIs	Different ways to work with the various bits of XML documents

While the DOM is low level and requires a lot of code to precisely formulate what you want to achieve, LINQ to XML provides a higher level syntax that allows doing simple things simply.

LINQ to XML also enables an *element-centric* approach in comparison to the *document-centric* approach of the DOM. This means that you can easily work with XML fragments (elements and attributes) without having to create a complete XML document.

Another API that the .NET Framework offers is the *XmlReader* and *XmlWriter* couple. These classes provide support for working on XML text in its raw form. This API is much more low-level than LINQ to XML. In fact, LINQ to XML uses the *XmlReader* and *XmlWriter* classes underneath and is not a completely new XML API without relations with previous APIs. On advantage of this is that is allows LINQ to XML to remain compatible with *XmlReader* and *XmlWriter*.

LINQ to XML makes creating documents more direct, but it also makes it easier to query XML documents. Expressing queries against XML documents feels more natural than having to write of lot of code with several loop instructions. Also, being part of the LINQ family of technologies, it is a good choice when you need to join across diverse data sources, be they XML documents, relational databases or objects.

With LINQ to XML, Microsoft is aiming at 80 percent of the use cases. These cases involve straightforward XML formats and common processing. For the other cases, developers will continue to use the other APIs. Also, while LINQ to XML takes inspiration from XSLT, XPath and XQuery, it is in no way able to compete with them. These technologies have benefits of their own and are designed for specific use cases. LINQ to XML is designed to make things easier for the vast majority of simple needs. If LINQ to XML is not enough for some specific cases, its compatibility with the other XML APIs allows making use of it in combination with these APIs. We'll keep this kind of advanced scenarios for Part 4 of this book.

For the moment, let's discover how LINQ to XML makes a difference by taking a look at some code.

1.5.2 Hello LINQ to XML

The running example application we'll use in this book deals with books. We'll detail this example in chapter 4. For the moment we'll stick to a simple *Book* class because it is enough for your first contact with LINQ to XML.

In our first LINQ to XML example, we want to filter and save a set of *Book* objects as XML. Here is how the *Book* class could be defined in C#[8]:

C#

```
class Book
{
  public string Publisher;
  public string Title;
  public int    Year;

  public Book(string title, string publisher, int year)
  {
    Title = title;
    Publisher = publisher;
    Year = year;
  }
}
```

and in VB.NET:

VB

```
Public Class Book
  Public Publisher As String
  Public Title As String
  Public Year As Integer

  Public Sub New(ByVal title As String, ByVal publisher As String, _
      ByVal year As Integer)
    Me.Title = title
    Me.Publisher = publisher
    Me.Year = year
  End Sub
End Class
```

Let say we have the following collection of books:

```
Book[] books = new Book[] {
  new Book("Ajax in Action", "Manning", 2005),
  new Book("Windows Forms in Action", "Manning", 2006),
  new Book("ASP.NET 2.0 Web Parts in Action", "Manning", 2006)
};
```

[8] Here we use public fields in the *Book* class for the sake of simplicity, but properties and private fields would be better. Another option is to use Auto-Implemented Properties, which is a new feature of C# 3.5. You'll be able to see Auto-Implemented Properties in action in chapters 7 and 13.

Here is the result we would like to get if we ask for the books published in 2006:

```
<books>
  <book title="Windows Forms in Action">
    <publisher>Manning</publisher>
  </book>
  <book title="ASP.NET 2.0 Web Parts in Action">
    <publisher>Manning</publisher>
  </book>
</books>
```

Using LINQ to XML, this can be done with the following code:

Listing 1.10 Hello LINQ to XML in C#

C#

```
using System;
using System.Linq;
using System.Xml;
using System.Xml.Linq;

class Book
{
  public string Publisher;
  public string Title;
  public int    Year;

  public Book(string title, string publisher, int year)
  {
    Title = title;
    Publisher = publisher;
    Year = year;
  }
}

static class HelloLinqToXml
{
  static void Main()
  {
    // Our book collection
    Book[] books = new Book[] {
      new Book("Ajax in Action", "Manning", 2005),
      new Book("Windows Forms in Action", "Manning", 2006),
      new Book("ASP.NET 2.0 Web Parts in Action", "Manning", 2006)
    };

    // Build the XML fragment based on the collection
    XElement xml = new XElement("books",
      from book in books
      where book.Year == 2006
      select new XElement("book",
        new XAttribute("title", book.Title),
        new XElement("publisher", book.Publisher)
      )
```

```
    );

    // Dump the XML to the console
    Console.WriteLine(xml);
  }
}
```

The same code in VB.NET:

Listing 1.11 Hello LINQ to XML in VB.NET

VB

```
Module HelloLinqToXml

  Public Class Book
    Public Publisher As String
    Public Title As String
    Public Year As Integer

    Public Sub New(ByVal title As String, ByVal publisher As String, _
        ByVal year As Integer)
      Me.Title = title
      Me.Publisher = publisher
      Me.Year = year
    End Sub
  End Class

  Sub Main()
    ' Our book collection
    Dim books As Book() = { _
      New Book("Ajax in Action", "Manning", 2005), _
      New Book("Windows Forms in Action", "Manning", 2006), _
      New Book("ASP.NET 2.0 Web Parts in Action", "Manning", 2006) _
    }

    ' Build the XML fragment based on the collection
    Dim xml As XElement = New XElement("books", _
      From book In books _
      Where book.Year = 2006 _
      Select New XElement("book", _
        New XAttribute("title", book.Title), _
        New XElement("publisher", book.Publisher) _
      ) _
    )

    ' Dump the XML to the console
    Console.WriteLine(xml)
  End Sub

End Module
```

In contrast, here is how we would build the same document without LINQ to XML, using the XML DOM:

Listing 1.12 Old-school version of Hello LINQ to XML

C#

```
using System;
using System.Xml;

class Book
{
  public string Title;
  public string Publisher;
  public int    Year;

  public Book(string title, string publisher, int year)
  {
    Title = title;
    Publisher = publisher;
    Year = year;
  }
}

static class HelloLinqToXml
{
  static void Main()
  {
    // Our book collection
    Book[] books = new Book[] {
      new Book("Ajax in Action", "Manning", 2005),
      new Book("Windows Forms in Action", "Manning", 2006),
      new Book("ASP.NET 2.0 Web Parts in Action", "Manning", 2006)
    };

    // Build the XML fragment based on the collection
    XmlDocument doc = new XmlDocument();
    XmlElement root = doc.CreateElement("books");
    foreach (Book book in books)
    {
      if (book.Year == 2006)
      {
        XmlElement element = doc.CreateElement("book");
        element.SetAttribute("title", book.Title);

        XmlElement publisher = doc.CreateElement("publisher");
        publisher.InnerText = book.Publisher;
        element.AppendChild(publisher);

        root.AppendChild(element);
      }
    }
    doc.AppendChild(root);

    // Display the result XML
    doc.Save(Console.Out);
  }
}
```

As you can see, LINQ to XML is more "visual" than the DOM: The structure of the code to get our XML fragment is close to the document we want to produce itself. We could say that it's WYSIWYM code: What You See Is What You Mean.

Microsoft names this approach the *Functional Construction* pattern. It allows you to structure your code in such a way that it reflects the shape of the XML document (or fragment) that you are constructing.

In VB.NET, the code can be even closer to the resulting XML:

Listing 1.13 Hello LINQ to XML VB.NET using XML literals

VB

```
Module XmlLiterals

  Sub Main()
    ' Our book collection
    Dim books as Book() = { _
      New Book("Ajax in Action", "Manning", 2005), _
      New Book("Windows Forms in Action", "Manning", 2006), _
      New Book("ASP.NET 2.0 Web Parts in Action", "Manning", 2006) _
    }

    ' Build an XML fragment using XML literals
    Dim xml As XElement = _
      <books>
        <%= From book In books _
            Where book.Year = 2006 _
            Select _
              <book title=<%= book.Title %>>
                <publisher><%= book.Publisher %></publisher>
              </book> _
        %>
      </books>

    ' Display the result XML
    Console.WriteLine(xml)
  End Sub

End Module
```

The above code uses a new syntax named *XML Literals*. *Literal* designates something that is output as is as part of the result. Here, the *books*, *book* and *publisher* XML elements will be part of the generated XML. XML Literals allow using a template of the XML we'd like to get, with a syntax those of you who do web development can compare to ASP.

The XML literals feature is not provided by C# 3.0. It exists only in VB.NET 9.0. You will discover that VB.NET comes with more language-integrated features than C# to work with XML.

You'll get the details about XML literals and everything else you need to know to make the best of LINQ to XML in part 4 of the book. For the moment, we still have one major piece of the LINQ trilogy to introduce: LINQ to SQL.

1.6 First steps with LINQ to SQL: Querying relational databases

LINQ's ambition is to make queries a natural part of the programming language. LINQ to SQL, which made its first appearance as DLinq, applies this concept to allow developers to query relational database using the same syntax that we have seen with LINQ to Objects and LINQ to XML.

After summing up how LINQ to SQL will help us, we'll show you how to write your first LINQ to SQL code.

1.6.1 What makes LINQ to SQL special

LINQ to SQL provides language-integrated data access by using LINQ's extension mechanism. It builds on ADO.NET to map tables and rows to classes and objects.

LINQ to SQL uses mapping information encoded in .NET custom attributes or contained in an XML document. This information is used to automatically handle the persistence of objects in relational databases. A table can be mapped to a class, the table's columns to properties of the class, and relationships between tables can be represented by properties.

LINQ to SQL automatically keeps track of changes on objects and updates the database accordingly through dynamic SQL queries or stored procedures. This is why you won't have to provide the SQL queries by yourself most of the time. But all this will be developed in Part 3 of this book. For the moment, let's make our first steps with LINQ to SQL code.

1.6.2 Hello LINQ to SQL

The time has come to take a look at some code using LINQ to SQL. As you have seen in our Hello LINQ example, we are able to write queries against a collection of objects. The following C# code snippet filters an in-memory collection of contacts based on their city:

```
from contact in contacts
where contact.City == "Paris"
select contact;
```

The good news is that thanks to LINQ to SQL, doing the same on data coming from a relational database is direct:

```
from contact in db.GetTable<Contact>()
where contact.City == "Paris"
select contact;
```

This query works on a list of contacts from a database. Notice how subtle the difference is between the two queries! In fact, only the object on which we are working is different, the query syntax is exactly the same. This shows how we'll be able to work the same way with multiple kind of data. This is what is so great about LINQ!

As an astute reader, you know that the language a relational database understands is SQL and you suspect that our LINQ query must be translated into a SQL query at some point. This is really the heart of the

technology: in the first example the collection is iterated in memory, while in the second code snippet the query is used to generate a SQL query that is sent to a database server. In the case of LINQ to SQL queries, the real processing happens on the database server. What's appealing about these queries is that we have a nice strongly-typed query API, in contrast with SQL where queries are expressed in strings and not validated at compile-time.

We will dissect the inner workings of LINQ to SQL in part 3 of this book, but let's first walk through a simple complete example. To begin with, I guess you're wondering what *db.GetTable<Contact>()* means in the LINQ to SQL query above. Keep reading and you'll discover the answer in the next section.

Entity classes

The first step in building a LINQ to SQL application is declaring the classes you'll use to represent your application data: your entities.

In our simple example, we'll define a class named *Contact* and associate it with the *Contacts* table of the Northwind sample database provided with LINQ. To do this, we need only to apply a custom attribute to the class:

```
[Table(Name="Contacts")]
class Contact
{
  public int ContactID;
  public string Name;
  public string City;
}
```

The Table attribute is provided by LINQ to SQL. It has a Name property that is used to specify the name of the database table.

In addition to associating entity classes to tables, we need to denote each field or property we intend to associate with a column of the table. This is done with the Column attribute.

```
[Table(Name="Contacts")]
class Contact
{
  [Column(Id=true)]
  public string ContactID;
  [Column("ContactName"]
  public string Name;
  [Column]
  public string City;
}
```

The Column attribute has a variety of properties you can use to customize the exact mapping between your fields or properties and the database's columns. You can see that we use the Id property to tell LINQ to SQL that the table column named ContactID is part of the table's primary key. Notice how we indicate that the ContactName column is to be mapped to the Name field. We don't specify name of the other columns or the types of the columns: in our case LINQ to SQL will deduce them from the class fields.

The DataContext

The next thing we need to prepare before being able to use language-integrated queries is a *System.Data.Linq.DataContext* object. The purpose of *DataContext* is to translate requests for objects into SQL queries made against the database and then assemble objects out of the results.

We will use the Northwnd.mdf database provided with the code samples accompanying this book. This database is the *Data* directory, so the creation of the *DataContext* object looks like this:

```
string dbPath = Path.GetFullPath(@"..\..\..\..\Data\northwnd.mdf");
DataContext db = new DataContext(dbPath);
```

The constructor of the *DataContext* class takes a connection string as a parameter. Since we are using SQL Server 2005 Express Edition, a path to the database file is sufficient.

The *DataContext* provides access to the tables in the database. Here is how to get access to the *Contacts* table mapped to our *Contact* class:

```
Table<Contact> contacts = db.GetTable<Contact>();
```

DataContext.GetTable is a generic method, which allows working with strongly-typed objects. This is what will allow us to use a LINQ query.

We are now able to write a complete code sample:

Listing 1.14 Hello LINQ to SQL complete source code

```
using System;
using System.Linq;
using System.Data.Linq;

static class HelloLinqToSql
{
  [Table(Name="Contacts")]
  class Contact
  {
    [Column(IsPrimaryKey=true)]
    public int ContactID;
    [Column(Name="ContactName")]
    public string Name;
    [Column]
    public string City;
  }

  static void Main()
  {
    // Get access to the database
    string dbPath = Path.GetFullPath(@"..\..\..\..\Data\northwnd.mdf");
    DataContext db = new DataContext(dbPath);

    // Query for contacts from Paris
    var contacts =
      from contact in db.GetTable<Contact>()
      where contact.City == "Paris"
```

```
      select contact;

    // Display the list of matching contacts
    foreach (var contact in contacts)
      Console.WriteLine("Bonjour "+contact.Name);
  }
}
```

Executing this code gives the following result:

```
Bonjour Marie Bertrand
Bonjour Dominique Perrier
Bonjour Guylène Nodier
```

Here is the SQL query that was sent to the server transparently:

```
SELECT [t0].[ContactID], [t0].[ContactName] AS [Name], [t0].[City]
FROM [Contacts] AS [t0]
WHERE [t0].[City] = @p0
```

Did you see how easy it was to get strongly-typed access to a database thanks to LINQ? Of course, this was a simplistic example, but hopefully it gives you a good idea of what LINQ to SQL has to offer and how it could change the way you work with databases.

Let's sum up what has been done automatically for us by LINQ to SQL:

- Opening a connection to the database
- Generating the SQL query
- Executing the SQL query against the database
- Creating and filled our objects out of the tabular results

As an exercise, you could try to do the same without LINQ to SQL. For example, you can try to use a *DataReader*. You would notice the following things in the *old-school* code when comparing it with our LINQ to SQL code:

- Queries explicitly written SQL in quotes
- No compile-time checks
- Loosely-bound parameters
- Loosely-typed result sets
- More code required
- More knowledge required

Writing standard data-access code hinders productivity for simple cases. In contrast, LINQ to SQL allows writing data-access code that doesn't get in the way.

Before concluding our introduction to LINQ to SQL, let's review some of its features.

1.6.3 A closer look at LINQ to SQL

Although LINQ to SQL is not a complete object-relational mapper and cannot compete with some of the existing mapping products available on the market, it still comes with an interesting set of features. Include a word about the ADO.NET Entity Framework. We will review these features in the book, but let me give you an idea of what you can expect.

You have seen that LINQ to SQL is able to generate dynamic SQL queries based on your language-integrated queries. This may not be adapted to every situation and so LINQ to SQL also supports custom SQL queries and stored procedures so that you can use your own hand-written SQL code and still benefit from the LINQ to SQL infrastructure.

In our example, we have provided the mapping information using custom attributes on our classes, but if you prefer not to have this kind of information hard-coded in your binaries, you are free to use an external XML mapping file to do the same.

To get a better understanding of how LINQ to SQL works, we have created our entity classes and provided the mapping information by ourselves. Most of the time, this code could be generated by tools coming with LINQ to SQL or using the graphical LINQ to SQL Designer.

The list of LINQ to SQL's features is much longer than this and also includes various things like: support for data-binding, interoperability with ADO.NET, concurrency management, support for inheritance, and help for debugging. Let's keep that for later, we promise that all this and even more will be covered in detail in part 3 of the book.

1.7 Summary

This first chapter presented the motivation behind the LINQ technologies. You also took your first steps with LINQ to Objects, LINQ to XML and LINQ to SQL code. Of course, we have just scratched the surface of the possibilities offered by LINQ et al., but hopefully you have been able to get an overview of some of the services these technologies provide. As you've seen, LINQ is not about taking SQL or XML and slapping them into C# or VB.NET code. It's much more than that as you'll see soon in the next chapters!

We hope that this introduction has stirred up your interest! But since LINQ is rich, we have a few things to cover before you can be on the verge of using it extensively in your applications. We will start in the next chapter by reviewing the enhancements that have been made to the C# and VB.NET languages to enable language-integrated queries.