LINQ in Action

SAMPLE CHAPTER

Fabrice Marguerie
Steve Eichert
Jim Wooley

FOREWORD BY Matt Warren
Principal Architect, Microsoft

MANNING

LINQ in Action

Fabrice Marguerie
Steve Eichert
Jim Wooley

Chapter 3

brief contents

PART 1 GETTING STARTED 1

1 ■ Introducing LINQ 3
2 ■ C# and VB.NET language enhancements 44
3 ■ LINQ building blocks 82

PART 2 QUERYING OBJECTS IN MEMORY 113

4 ■ Getting familiar with LINQ to Objects 115
5 ■ Beyond basic in-memory queries 160

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

6 ■ Getting started with LINQ to SQL 205
7 ■ Peeking under the covers of LINQ to SQL 237
8 ■ Advanced LINQ to SQL features 267

PART 4 MANIPULATING XML 311

9 ■ Introducing LINQ to XML 313
10 ■ Query and transform XML with LINQ to XML 350
11 ■ Common LINQ to XML Scenarios 385

PART 5 LINQING IT ALL TOGETHER 435

12 ▪ Extending LINQ 437

13 ▪ LINQ in every layer 482

3 LINQ building blocks

This chapter covers:

- An introduction to the key elements of the LINQ foundation
- Sequences
- Deferred query execution
- Query operators
- Query expressions
- Expression trees
- LINQ DLLs and namespaces

In chapter 2, we reviewed the language additions made to C# and VB.NET: the basic elements and language innovations that make LINQ possible.

In this chapter, you'll discover new concepts unique to LINQ. Each of these concepts builds on the new language features we presented in chapter 2. You'll now begin to see how everything adds up when used by LINQ.

We'll start with a rundown of the language features we've already covered. We'll then present new features that form the key elements of the LINQ foundation. In particular, we'll detail the language extensions and key concepts. This includes sequences, the standard query operators, query expressions, and expression trees. We'll finish this chapter by taking a look at how LINQ extends the .NET Framework with new assemblies and namespaces.

At the end of this chapter, you should have a good overview of all the fundamental building blocks on which LINQ relies and how they fit together. With this foundation, you'll be ready to work on LINQ code.

3.1 How LINQ extends .NET

This section gives a refresher on the features we introduced in chapter 2 and puts them into the big picture so you can get a clear idea of how they all work together when used with LINQ. We'll also enumerate the elements LINQ brings to the party, which we'll detail in the rest of this chapter.

3.1.1 Refresher on the language extensions

As a refresher, let's sum up the significant additions to the languages that you discovered in chapter 2:

- Implicitly typed local variables
- Object initializers
- Lambda expressions
- Extension methods
- Anonymous types

These additions are what we call *language extensions*, the set of new language features and syntactic constructs added to C# and VB.NET to support LINQ. All of these extensions require new versions of the C# and VB.NET compilers, but no new IL instructions or changes of the .NET runtime.

These language extensions are full-fledged features that can be used in code that has nothing to do with LINQ. They are however required for LINQ to work, and you'll use them a lot when writing language-integrated queries.

In order to introduce LINQ concepts and understand why they are important, we'll dissect a code sample throughout this chapter. We'll keep the same subject as in chapter 2: filtering and sorting a list of running processes.

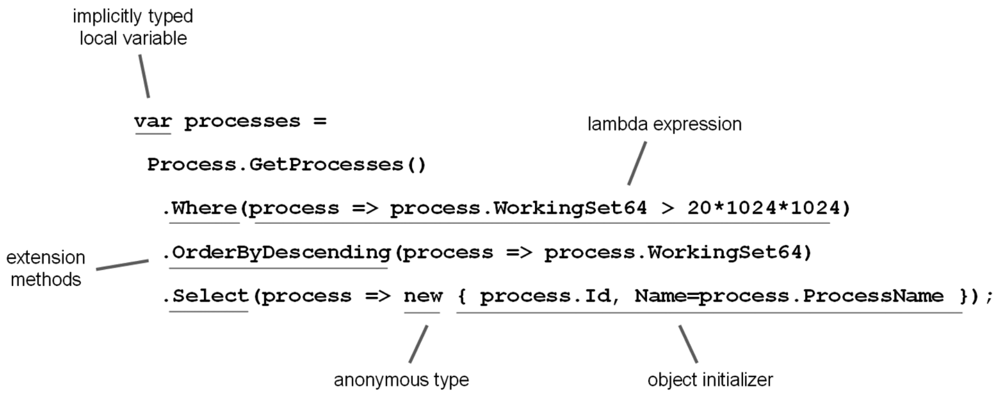
Here is the code sample we'll use:

```
static void DisplayProcesses()
{
  var processes =
    Process.GetProcesses()
      .Where(process => process.WorkingSet64 > 20*1024*1024)
      .OrderByDescending(process => process.WorkingSet64)
      .Select(process => new { process.Id,
                               Name=process.ProcessName });

  ObjectDumper.Write(processes);
}
```

The portion of code in bold is a LINQ query. If you take a close look at it, you can see all the language enhancements we introduced in the previous chapter, as shown in figure 3.1.

In the figure, you should clearly see how everything dovetails to form a complete solution. You can now understand why we called the language enhancements "key components" for LINQ.



```
var processes =
 Process.GetProcesses()
  .Where(process => process.WorkingSet64 > 20*1024*1024)
  .OrderByDescending(process => process.WorkingSet64)
  .Select(process => new { process.Id, Name=process.ProcessName });
```

Figure 3.1 The language extensions all in one picture

3.1.2 The key elements of the LINQ foundation

More features and concepts are required for LINQ to work than those we've just listed. Several concepts specifically related to queries are also required:

- We'll start by explaining what *sequences* are and how they are used in LINQ queries.
- You'll also encounter *query expressions*. This is the name for the `from…where…select` syntax you've already seen.
- We'll explore *query operators*, which represent the basic operations you can perform in a LINQ query.
- We'll also explain what *deferred query execution* means, and why it is important.
- In order to enable deferred query execution, LINQ uses *expression trees*. We'll see what expression trees are and how LINQ uses them.

You need to understand these features in order to be able to read and write LINQ code, as we'll do in the next chapters.

3.2 Introducing sequences

The first LINQ concept we'll present in this chapter is the sequence.

In order to introduce sequences and understand why they are important, let's dissect listing 3.1.

Listing 3.1 Querying a list of processes using extension methods

```
var processes =
  Process.GetProcesses()    ❶
    .Where(process => process.WorkingSet64 > 20*1024*1024)    ❷
    .OrderByDescending(process => process.WorkingSet64)    ❸
    .Select(process => new { process.Id,
                             Name=process.ProcessName });    ❹
```

❶ Get a list of running processes

❷ Filter the list

❸ Sort the list

❹ Keep only the IDs and names

To precisely understand what happens under the covers, let's analyze this code step by step, in the order the processing happens.

We'll start by looking at `IEnumerable<T>`, a key interface you'll find everywhere when working with LINQ. We'll also provide a small refresher on *iterators* and then stress how iterators allow *deferred query execution*.

3.2.1 IEnumerable<T>

The first thing you need to understand in listing 3.1 is what the call to `Process.GetProcesses` ❶ returns and how it is used. The `GetProcesses` method of the `System.Diagnostics.Process` class returns an array of `Process` objects. This is not surprising and probably wouldn't be interesting, except that arrays implement the generic `IEnumerable<T>` interface. This interface, which appeared with .NET 2.0, is key to LINQ. In our particular case, an array of `Process` objects implements `IEnumerable<Process>`.

The `IEnumerable<T>` interface is important because `Where` ❷, `OrderByDescending` ❸, `Select` ❹, and other standard query operators used in LINQ queries expect an object of this type as a parameter.

Listing 3.2 shows how the `Where` method is defined, for instance.

Listing 3.2 The `Where` method that is used in our sample query

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,     ❶
  Func<TSource, Boolean> predicate)
{
  foreach (TSource element in source)
  {
    if (predicate(element))
      yield return element;     ❷
  }
}
```

But where does this `Where` method come from? Is it a method of the `IEnumerable<T>` interface? Well, no. As you may have guessed if you remember chapter 2, it's an *extension method*. This can be detected by the presence of the `this` keyword on the first parameter of the method ❶.

The extension methods we see here (`Where`, `OrderByDescending`, and `Select`) are provided by the `System.Linq.Enumerable` class. The name of this class comes from the fact that the extension methods it contains work on `IEnumerable<T>` objects.

NOTE In LINQ, the term *sequence* designates everything that implements `IEnumerable<T>`.

Let's take another look at the `Where` method. Note that it uses the `yield return` ❷ statement added in C# 2.0. This and the `IEnumerable<TSource>` return type in the signature make it an iterator.

We'll now take some time to review background information on iterators before getting back to our example.

3.2.2 Refresher on iterators

An *iterator* is an object that allows you to traverse through a collection's elements. What is named an iterator in .NET is also known as a *generator* in other languages such as Python, or sometimes a *cursor*, especially within the context of a database.

You may not know what an iterator is, but you surely have used several of them before! Each time you use a `foreach` loop (`For Each` in VB.NET), an iterator is involved. (This isn't true for arrays because the C# and VB.NET compilers optimize `foreach` and `For Each` loops over arrays to replace the use of iterators by a simple loop, as if a `for` loop were used.) Every .NET collection (`List<T>`, `Dictionary<T>`, and `ArrayList` for example) has a method named `GetEnumerator` that returns an object used to iterate over its contents. That's what `foreach` uses behind the scenes to iterate on the items contained in a collection.

If you're interested in design patterns, you can study the classical Iterator pattern. This is the design iterators rely on in .NET.

An iterator is similar, in its result, to a traditional method that returns a collection, because it generates a sequence of values. For example, we could create the following method to return an enumeration of integers:

```
int[] OneTwoThree()
{
  return new [] {1, 2, 3};
}
```

However, the behavior of an iterator in C# 2.0 or 3.0 is very specific. Instead of building a collection containing all the values and returning them all at once, an iterator returns the values one at a time. This requires less memory and allows the caller to start processing the first few values immediately, without having the complete collection ready.

Let's look at a sample iterator to understand how it works. An iterator is easy to create: it's simply a method that returns an enumeration and uses `yield return` to provide the values.

Listing 3,3 shows an iterator named `OneTwoThree` that returns an enumeration containing the integer values 1, 2, and 3:

Listing 3.3 Sample iterator (Iterator.csproj)

```
using System;
using System.Collections.Generic;

static class Iterator
{
  static IEnumerable<int> OneTwoThree()
  {
    Console.WriteLine("Returning 1");
    yield return 1;
    Console.WriteLine("Returning 2");
    yield return 2;
    Console.WriteLine("Returning 3");
    yield return 3;
  }

  static void Main()
  {
    foreach (var number in OneTwoThree())
    {
      Console.WriteLine(number);
    }
  }
```

Here are the results of this code sample's execution:

```
Returning 1
1
Returning 2
2
Returning 3
3
```

As you can see, the `OneTwoThree` method does not exit until we reach its last statement. Each time we reach a `yield return` statement, the control is yielded back to the caller method. In our case, the `foreach` loop does its work, and then control is returned to the iterator method where it left so it can provide the next item.

It looks like two methods, or *routines*, are running at the same time. This is why .NET iterators could be presented as a kind of lightweight coroutine. A traditional method starts its execution at the beginning of its body each time it is called. This kind of method is named a *subroutine*. In comparison, a *coroutine* is a method that resumes its execution at the point it stopped the last time it was called, as if nothing had happened between invocations. All C# methods are subroutines except methods that contain a `yield return` instruction, which can be considered to be coroutines.[1]

One thing you may find strange is that although we implement a method that returns an `IEnumerable<int>` in listing 3.3, in appearance we don't return an object of that type. We use `yield return`. The compiler does the work for us, and a class implementing `IEnumerable<int>` is created *automagically* for us. The `yield return` keyword is a time-saver that instructs the compiler to create a state engine in IL so you can create methods that retain their state without having to go through the pain of maintaining state in your own code.

We won't go into more details on this subject in this book, because it's not required to understand LINQ, and anyway, this is a standard C# 2.0 feature. However, if you want to investigate this, .NET Reflector is your friend.[2]

NOTE VB.NET has no instruction equivalent to `yield return`. Without this shortcut, VB.NET developers have to implement the `IEnumerable(Of T)` interface by hand to create enumerators. We provide a sample implementation in the companion source code download. See the `Iterator.vbproj` project.

The simple example provided in listing 3.3 shows that iterators are based on lazy evaluation. We'd like to stress that this big characteristic of iterators is essential for LINQ, as you'll see next.

3.2.3 *Deferred query execution*

LINQ queries rely heavily on lazy evaluation. In LINQ vocabulary, we'll refer to this as *deferred query execution*, also called *deferred query evaluation*. This is one of the most important concepts in LINQ. Without this facility, LINQ would perform very poorly.

Let's take a simple example to demonstrate how a query execution behaves.

[1] See Patrick Smacchia's book *Practical .NET2 and C#2* (Paradoxal Press) if you want to learn more about iterators.

[2] If you want to look into the low-level machinery of how state engines are built to make iterators work in .NET, you can download .NET Reflector at http://aisto.com/roeder/dotnet.

Demonstrating deferred query execution

In listing 3.4, we'll query an array of integers and perform an operation on all the items it contains.

Listing 3.4 Deferred query execution demonstration (DeferredQueryExecution.csproj)

```
using System;
using System.Linq;

static class DeferredQueryExecution
{
  static double Square(double n)
  {
    Console.WriteLine("Computing Square("+n+")...");
    return Math.Pow(n, 2);
  }

  public static void Main()
  {
    int[] numbers = {1, 2, 3};

    var query =
      from n in numbers
      select Square(n);

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

The results of this program clearly show that the query does not execute at once. Instead, the query evaluates as we iterate on it:

```
Computing Square(1)...
1
Computing Square(2)...
4
Computing Square(3)...
9
```

As you'll see soon in section 3.4, queries such as the following one are translated into method calls at compile-time:

```
var query =
  from n in numbers
  select Square(n);
```

Once compiled, this query becomes

```
IEnumerable<double> query =
  Enumerable.Select<int, double>(numbers, n => Square(n));
```

The fact that the `Enumerable.Select` method is an iterator explains why we get delayed execution.

It is important to realize that our query variable represents not the result of a query, but merely the *potential* to execute a query. The query is not executed when it is assigned to a variable. It executes afterward, step by step.

One advantage of deferred query evaluation is that it conserves resources. The gist of lazy evaluation is that the data source on which a query operates is not iterated until you iterate over the query's results. Let's suppose a query returns thousands of elements. If we decide after looking at the first element that we don't want to further process the results, these results won't be loaded in memory. This is because the results are provided as a sequence. If the results were contained in an array or list as is often the case in classical programming, they would all be loaded in memory, even if we didn't consume them.

Deferred query evaluation is also important because it allows us to define a query at one point and use it later, exactly when we want to, several times if needed.

Reusing a query to get different results

An important thing to understand is that if you iterate on the same query a second time, it can produce different results. An example of this behavior can be seen in listing 3.5. New code is shown in bold.

Listing 3.5 Same query producing different results between two executions

```
using System;
using System.Linq;

static class QueryReuse
{
  static double Square(double n)
  {
    Console.WriteLine("Computing Square("+n+")...");
    return Math.Pow(n, 2);
  }

  public static void Main()
  {
    int[] numbers = {1, 2, 3};
```

```
      var query =
        from n in numbers
        select Square(n);

      foreach (var n in query)
        Console.WriteLine(n);

      for (int i = 0; i < numbers.Length; i++)
        numbers[i] = numbers[i]+10;

      Console.WriteLine("- Collection updated -");

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

Here we reuse the query object after changing the underlying collection. We add 10 to each number in the array before iterating again on the query.

As expected, the results are not the same for the second iteration:

```
Computing Square(1)...
1
Computing Square(2)...
4
Computing Square(3)...
9
- Collection updated -
Computing Square(11)...
121
Computing Square(12)...
144
Computing Square(13)...
169
```

The second iteration executes the query again, producing new results.

Forcing immediate query execution

As you've seen, deferred execution is the default behavior. Queries are executed only when we request data from them. If you want immediate execution, you have to request it explicitly.

Let's say that we want the query to be executed completely, before we begin to process its results. This would imply that all the calls to the `Square` method happen before the results are used.

Here is how the output should look without deferred execution:

```
Computing Square(1)...
Computing Square(2)...
Computing Square(3)...
1
4
9
```

We can achieve this by adding a call to `ToList`—another extension method from the `System.Linq.Enumerable` class—to our code sample:

```
foreach (var n in query.ToList())
  Console.WriteLine(n);
```

With this simple modification, our code's behavior changes radically.

`ToList` iterates on the query and creates an instance of `List<double>` initialized with all the results of the query. The `foreach` loop now iterates on a prefilled collection, and the `Square` method is not invoked during the iteration.

Let's go back to our `DisplayProcesses` example and continue analyzing the query.

The `Where`, `OrderByDescending`, and `Select` methods used in listing 3.1 are iterators. This means for example that the enumeration of the source sequence provided as the first parameter of a call to the `Where` method won't happen before we start enumerating the results. This is what allows delayed execution.

You'll now learn more about the extension methods provided by the `System.Linq.Enumerable` class.

3.3 Introducing query operators

We've used extension methods from the `System.Linq.Enumerable` class several times in our code samples. We'll now spend some time describing them more precisely. You'll learn how such methods, called query operators, are at the heart of the LINQ foundation. You should pay close attention to query operators, because you'll use them the most when writing LINQ queries.

We'll first define what a query operator is, before introducing the standard query operators.

3.3.1 What makes a query operator?

Query operators are not a language extension per se, but an extension to the .NET Framework Class Library. Query operators are a set of extension methods that perform operations in the context of LINQ queries. They are the real elements that make LINQ possible.

Before spending some time on iterators, we were looking at the `Where` method that is used in the following code sample:

```
var processes =
  Process.GetProcesses()
    .Where(process => process.WorkingSet64 > 20*1024*1024)      ❶ Call to Where
    .OrderByDescending(process => process.WorkingSet64)
    .Select(process => new { process.Id,
                             Name=process.ProcessName });
```

Let's take a deeper look at the `Where` method and analyze how it works. This method is provided by the `System.Linq.Enumerable` class. Here again is how it's implemented, as we showed in listing 3.2:

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  Func<TSource, Boolean> predicate)
{
  foreach (TSource element in source)          ❷ foreach loop
  {
    if (predicate(element))                    ❸ Filter source
      yield return element;                    ❹ Return elements
  }
}
```

Note that the `Where` method takes an `IEnumerable<T>` as an argument. This is not surprising, because it's an extension method that gets applied to the result of the call to `Process.GetProcesses`, which returns an `IEnumerable<Process>` as we've seen before. What is particularly interesting at this point is that the `Where` method also returns an `IEnumerable<T>`, or more precisely an `IEnumerable<Process>` in this context.

Here is how the `Where` method works:

❶ It is called with the list of processes returned by `Process.GetProcesses`.

❷ It loops on the list of processes it receives.

❸ It filters this list of processes.

❹ It returns the filtered list element by element.

Although we present the processing as four steps, you already know that the processes are handled one by one thanks to the use of `yield return` and iterators.

If we tell you that `OrderByDescending` and `Select` also take `IEnumerable<T>` and return `IEnumerable<T>`, you should start to see a pattern. `Where`, `OrderByDescending`, and `Select` are used in turn to refine the processing on the original enumeration. These methods operate on enumerations and generate enumerations. This looks like a Pipeline pattern, don't you think?

Do you remember how we said in chapter 2 that extension methods are basically static methods that can facilitate a chaining or pipelining pattern? If we remove the dot notation from this code snippet

```
var processes =
  Process.GetProcesses()
    .Where(process => process.WorkingSet64 > 20*1024*1024)
    .OrderByDescending(process => process.WorkingSet64)
    .Select(process => new { process.Id,
                             Name=process.ProcessName });
```

and transform it to use standard static method calls, it becomes listing 3.6.

Listing 3.6 A query expressed as static method calls

```
var processes =
  Enumerable.Select(
    Enumerable.OrderByDescending(
      Enumerable.Where(
        Process.GetProcesses(),
        process => process.WorkingSet64 > 20*1024*1024),
      process => process.WorkingSet64),
    process => new { process.Id, Name=process.ProcessName });
```

Again, you can see how extension methods make this kind of code much easier to read! If you look at the code sample that doesn't use extension methods, you can see how difficult it is to understand that we start the processing with a list of processes. It's also hard to follow how the method calls are chained to refine the results. It is in cases like this one that extension methods show all their power.

Until now in this chapter, we've stressed several characteristics of extension methods such as `Where`, `OrderByDescending`, and `Select`:

- They work on enumerations.
- They allow pipelined data processing.
- They rely on delayed execution.

All these features make these methods useful to write queries. This explains why these methods are called query operators.

Here is an interesting analogy. If we consider a query to be a factory, the query operators would be machines or engines, and sequences would be the material the query operators work on (see figure 3.2):

1 A sequence is provided at the start of the processing.

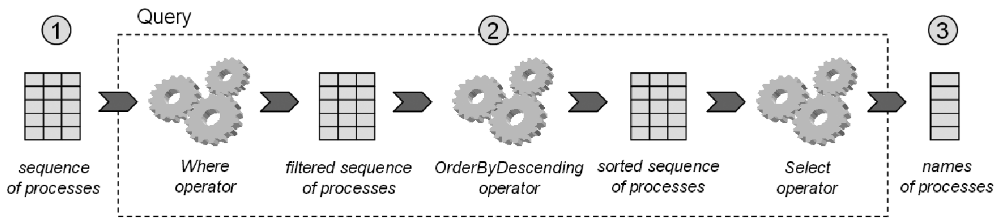

Query
1
2
3
sequence of processes
Where operator
filtered sequence of processes
OrderByDescending operator
sorted sequence of processes
Select operator
names of processes

Figure 3.2 A LINQ query represented as a factory where query operators are machines and sequences are the material.

2 Several operators are applied on the sequence to refine it.
3 The final sequence is the product of the query.

NOTE Don't be misled by figure 3.2. Each element in the sequence is processed only when it is requested. This is how delayed execution works. The elements in sequences are not processed in batch, and maybe even not all processed if not requested.

As we'll highlight in chapter 5, some intermediate operations (such as sorting and grouping) require the entire source be iterated over. Our `OrderByDescending` call is an example of this.

If we look at listing 3.6, we could say that queries are just made of a combination of query operators. Query operators are the key to LINQ, even more than language constructs like query expressions.

3.3.2 The standard query operators

Query operators can be combined to perform complex operations and queries on enumerations. Several query operators are predefined and cover a wide range of operations. These operators are called the *standard query operators.*

Table 3.1 classifies the standard query operators according to the type of operation they perform.

Table 3.1 The standard query operators grouped in families

Family	Query operators
Filtering	`OfType`, `Where`
Projection	`Select`, `SelectMany`
Partitioning	`Skip`, `SkipWhile`, `Take`, `TakeWhile`
Join	`GroupJoin`, `Join`

Table 3.1 The standard query operators grouped in families *(continued)*

Family	Query operators
Concatenation	`Concat`
Ordering	`OrderBy`, `OrderByDescending`, `Reverse`, `ThenBy`, `ThenByDescending`
Grouping	`GroupBy`, `ToLookup`
Set	`Distinct`, `Except`, `Intersect`, `Union`
Conversion	`AsEnumerable`, `AsQueryable`, `Cast`, `ToArray`, `ToDictionary`, `ToList`
Equality	`SequenceEqual`
Element	`ElementAt`, `ElementAtOrDefault`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`
Generation	`DefaultIfEmpty`, `Empty`, `Range`, `Repeat`
Quantifiers	`All`, `Any`, `Contains`
Aggregation	`Aggregate`, `Average`, `Count`, `LongCount`, `Max`, `Min`, `Sum`

As you can see, many operators are predefined. For reference, you can find this list augmented with a description of each operator in the appendix. You'll also learn more about the standard query operators in chapter 4, where we'll provide several examples using them. We'll then demonstrate how they can be used to do projections, aggregation, sorting, or grouping.

Thanks to the fact that query operators are mainly extension methods working with `IEnumerable<T>` objects, you can easily create your own query operators. We'll see how to create and use domain-specific query operators in chapter 12, which covers extensibility.

3.4 Introducing query expressions

Another key concept of LINQ is a new language extension. C# and VB.NET propose syntactic sugar for writing simpler query code in most cases.

Until now, in this chapter, we've used a syntax based on method calls for our code samples. This is one way to express queries. But most of the time when you look at code based on LINQ, you'll notice a different syntax: *query expressions.*

We'll explain what query expressions are and then describe the relationship between query expressions and query operators.

3.4.1 What is a query expression?

Query operators are static methods that allow the expression of queries. But instead of using the following syntax

```
var processes =
  Process.GetProcesses()
    .Where(process => process.WorkingSet64 > 20*1024*1024)
    .OrderByDescending(process => process.WorkingSet64)
    .Select(process => new { process.Id,
                             Name=process.ProcessName });
```

you can use another syntax that makes LINQ queries resemble SQL queries (see QueryExpression.csproj):

```
var processes =
  from process in Process.GetProcesses()
  where process.WorkingSet64 > 20*1024*1024
  orderby process.WorkingSet64 descending
  select new { process.Id, Name=process.ProcessName };
```

This is called a query expression or query syntax.

The two code pieces are semantically identical. A query expression is convenient declarative shorthand for code you could write manually. Query expressions allow us to use the power of query operators, but with a query-oriented syntax.

Query expressions provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery. A query expression operates on one or more information sources by applying one or more query operators from either the standard query operators or domain-specific operators. In our code sample, the query expression uses three of the standard query operators: `Where`, `OrderByDescending`, and `Select`.

When you use a query expression, the compiler automagically translates it into calls to standard query operators.

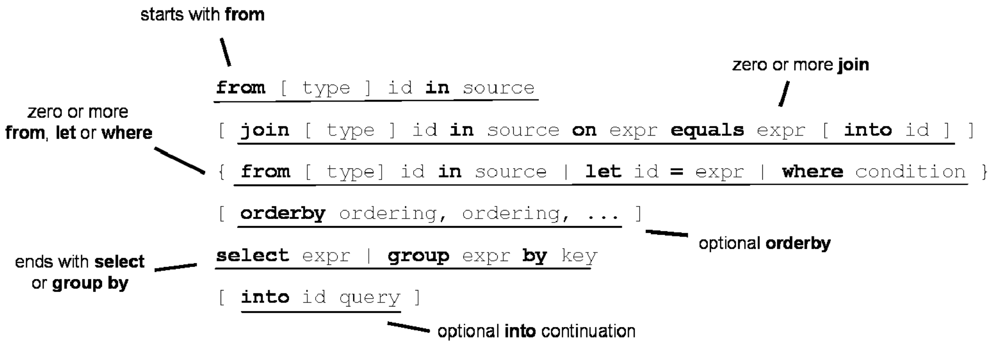
Because query expressions compile down to method calls, they are not necessary: We could work directly with the query operators. The big advantage of query expressions is that they allow for greater readability and simplicity.

3.4.2 Writing query expressions

Let's detail what query expressions look like in C# and in VB.NET.

C# syntax

Figure 3.3 shows the exhaustive syntax for a query expression.



```
from [ type ] id in source
[ join [ type ] id in source on expr equals expr [ into id ] ]
{ from [ type] id in source | let id = expr | where condition }
[ orderby ordering, ordering, ... ]
select expr | group expr by key
[ into id query ]
```

starts with **from**
zero or more **join**
zero or more **from**, **let** or **where**
optional **orderby**
ends with **select** or **group by**
optional **into** continuation

Figure 3.3 C# query expression syntax

Let's review how this syntax is presented in the C# 3.0 language specification. A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join`, or `orderby` clauses.

Each `from` clause is a generator introducing a variable that ranges over the elements of a sequence. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result.

Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria. The final `select` or `group` clause specifies the shape of the result in terms of the range variables.

Finally, an `into` clause can be used to splice queries by treating the results of one query as a generator in a subsequent query.

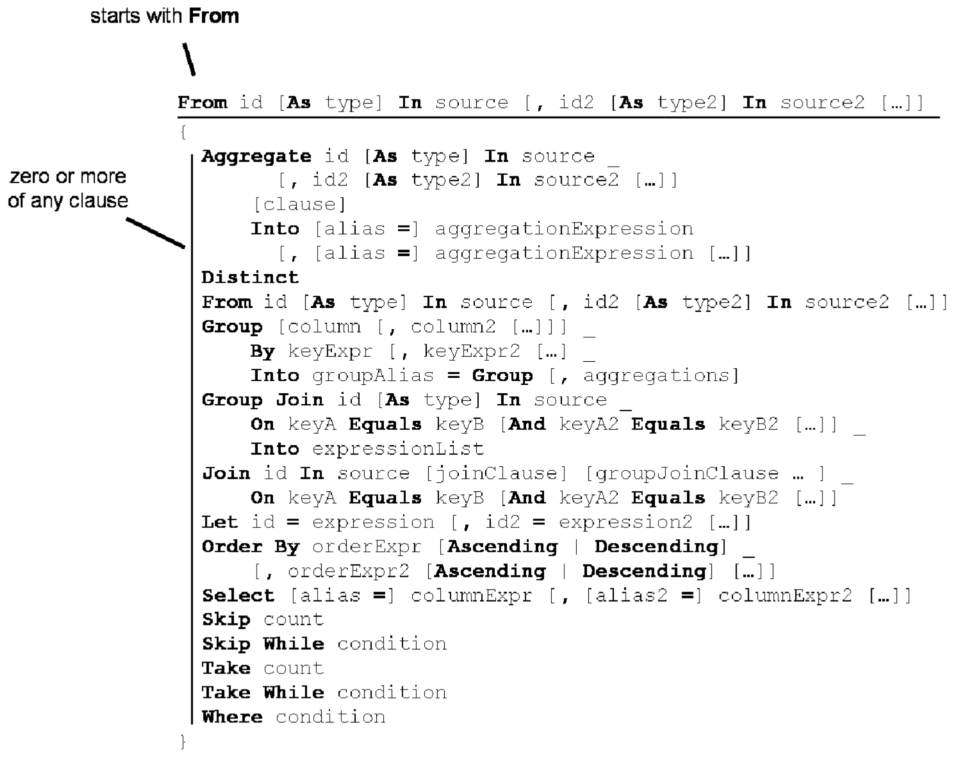
This syntax should not be unfamiliar if you know SQL.

VB.NET syntax

Figure 3.4 depicts the syntax of a query expression in VB.NET.

Notice how the VB.NET query expression syntax is richer compared to C#. More of the standard query operators are supported in VB, such as `Distinct`, `Skip`, `Take`, and the aggregation operators.

We'll use query expressions extensively in the rest of the book. We believe it's easier to discover the syntax through code samples instead of analyzing and exposing the exact syntax at this point. You'll see query expressions in action in chapter 4, for instance, where we'll use all kinds of queries. This will help you to



```
From id [As type] In source [, id2 [As type2] In source2 […]]
{
  Aggregate id [As type] In source _
        [, id2 [As type2] In source2 […]]
      [clause]
      Into [alias =] aggregationExpression
        [, [alias =] aggregationExpression […]]
  Distinct
  From id [As type] In source [, id2 [As type2] In source2 […]]
  Group [column [, column2 […]]] _
      By keyExpr [, keyExpr2 […] _
      Into groupAlias = Group [, aggregations]
  Group Join id [As type] In source _
      On keyA Equals keyB [And keyA2 Equals keyB2 […]] _
      Into expressionList
  Join id In source [joinClause] [groupJoinClause … ] _
      On keyA Equals keyB [And keyA2 Equals keyB2 […]]
  Let id = expression [, id2 = expression2 […]]
  Order By orderExpr [Ascending | Descending] _
      [, orderExpr2 [Ascending | Descending] […]]
  Select [alias =] columnExpr [, [alias2 =] columnExpr2 […]]
  Skip count
  Skip While condition
  Take count
  Take While condition
  Where condition
}
```

Figure 3.4 VB.NET query expression syntax

learn everything you need to use query expressions. In addition, Visual Studio's IntelliSense will help you to write query expressions and discover their syntax as you type them.

3.4.3 *How the standard query operators relate to query expressions*

You've seen that a translation happens when a query expression is compiled into calls to standard query operators.

For instance, consider our query expression:

```
from process in Process.GetProcesses()
where process.WorkingSet64 > 20*1024*1024
orderby process.WorkingSet64 descending
select new { process.Id, Name=process.ProcessName };
```

Here is the same query formulated with query operators:

```
Process.GetProcesses()
  .Where(process => process.WorkingSet64 > 20*1024*1024)
  .OrderByDescending(process => process.WorkingSet64)
  .Select(process => new { process.Id, Name=process.ProcessName });
```

Table 3.2 shows how the major standard query operators are mapped to the new C# and VB.NET query expression keywords.

Table 3.2 Mapping of standard query operators to query expression keywords by language

Query operator	C# syntax	VB.NET syntax
`All`	N/A	`Aggregate … In … Into All(…)`
`Any`	N/A	`Aggregate … In … Into Any()`
`Average`	N/A	`Aggregate … In … Into Average()`
`Cast`	Use an explicitly typed range variable, for example: `from int i in numbers`	`From … As …`
`Count`	N/A	`Aggregate … In … Into Count()`
`Distinct`	N/A	`Distinct`

`GroupBy`	`group … by`
	or
	`group … by … into …`
		`Group … By … Into …`

`GroupJoin`	`join … in … on … equals … into…`	`Group Join … In … On …`

`Join`	`join … in … on … equals …`	`From x In …, y In … Where x.a = b.a`
		or
		`Join … [As …] In … On …`

`LongCount`	N/A	`Aggregate … In … Into LongCount()`
`Max`	N/A	`Aggregate … In … Into Max()`
`Min`	N/A	`Aggregate … In … Into Min()`
`OrderBy`	`orderby`	`Order By`
`OrderByDescending`	`orderby … descending`	`Order By … Descending`
`Select`	`select`	`Select`
`SelectMany`	Multiple `from` clauses	Multiple `From` clauses
`Skip`	N/A	`Skip`
`SkipWhile`	N/A	`Skip While`

Table 3.2 Mapping of standard query operators to query expression keywords by language *(continued)*

Query operator	C# syntax	VB.NET syntax
`Sum`	`N/A`	`Aggregate … In … Into Sum()`
`Take`	`N/A`	`Take`
`TakeWhile`	`N/A`	`Take While`
`ThenBy`	`orderby …, …`	`Order By …, …`
`ThenByDescending`	`orderby …, … descending`	`Order By …, … Descending`
`Where`	`where`	`Where`

As you can see, not all operators have equivalent keywords in C# and VB.NET. In your simplest queries, you'll be able to use the keywords proposed by your programming language; but for advanced queries, you'll have to call the query operators directly, as you'll see in chapter 4.

Also, writing a query using a query expression is only for comfort and readability; in the end, once compiled, it gets converted into calls to standard query operators. You could decide to write all your queries only with query operators and avoid the query expression syntax if you prefer.

3.4.4 Limitations

Throughout this book, we'll write queries either using the query operators directly or using query expressions. Even when using query expressions, we may have to explicitly use some of the query operators. Only a subset of the standard query operators is supported by the query expression syntax and keywords. It's often necessary to work with some of the query operators right in the context of a query expression.

The C# compiler translates query expressions into invocations of the following operators: `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`, as shown in table 3.2. If you need to use other operators, you can do so in the context of a query expression.

For example, in listing 3.7, we use the `Take` and `Distinct` operators.

Listing 3.7 C# query expression that uses query operators (QueryExpressionWithOperators.csproj)

```
var authors =
  from distinctAuthor in (
    from book in SampleData.Books
    where book.Title.Contains("LINQ")
    from author in book.Authors.Take(1)
    select author)
    .Distinct()
  select new {distinctAuthor.FirstName, distinctAuthor.LastName};
```

NOTE `SampleData` is a class we'll define when we introduce our running example in chapter 4. It provides some sample data on books, authors, and publishers.

We use `Take` and `Distinct` explicitly. Other operators are used implicitly in this query, namely `Where`, `Select`, and `SelectMany`, which correspond to the `where`, `select`, and `from` keywords.

In listing 3.7, the query selects a list of the names of the first author of each book that contains "LINQ" in its title, a given author being listed only once.

Listing 3.8 shows how the same query can be written with query operators only.

Listing 3.8 C# query that uses query operators only (QueryExpressionWithOperators.csproj)

```
var authors =
  SampleData.Books
    .Where(book => book.Title.Contains("LINQ"))
    .SelectMany(book => book.Authors.Take(1))
    .Distinct()
    .Select(author => new {author.FirstName, author.LastName});
```

It's up to you to decide what's more readable. In some cases, you'll prefer to use a combination of query operators because a query expression wouldn't make things clearer. Sometimes, query expressions can even make code more difficult to understand.

In listing 3.7, you can see that parentheses are required to use the `Distinct` operator. This gets in the middle of the query expression and makes it more difficult to read. In listing 3.8, where only query operators are used, it's easier to follow the pipelined processing. The query operators allow us to organize the operations sequentially. Note that in VB, the question is less important because the language offers more keywords mapped to query operators. This includes `Take` and `Distinct`. Consequently, the query we've just written in C# can be written completely in VB as a query expression without resorting to query operators.

If you're used to working with SQL, you may also like query expressions because they offer a similar syntax. Another reason for preferring query expression is that they offer a more compact syntax than query operators.

Let's take the following queries for example. First, here is a query with query operators:

```
SampleData.Books
  .Where(book => book.Title == "Funny Stories")
  .OrderBy(book => book.Title)
  .Select(book => new {book.Title, book.Price});
```

Here is the same query with a query expression:

```
from book in SampleData.Books
where book.Title == "Funny Stories"
orderby book.Title
select new {book.Title, book.Price};
```

The two queries are equivalent. But you might notice that the query formulated with query operators makes extensive use of lambda expressions. Lambda expressions are useful, but too many in a small block of code can be unattractive. Also, in the same query, notice how the book identifier is declared several times. In comparison, in the query expression, you can see that the book identifier only needs to be declared once.

Again, it's mainly a question of personal preference, so we do not intend to tell you that one way is better than the other.

After query expressions, we have one last LINQ concept to introduce.

3.5 Introducing expression trees

You might not use expression trees as often as the other concepts we've reviewed so far, but they are an important part of LINQ. They allow advanced extensibility and make LINQ to SQL possible, for instance.

We'll spend some time again with lambda expressions because they allow us to create expression trees. We'll then detail what an expression tree is, before stressing how expression trees offer another way to enable deferred query execution.

3.5.1 *Return of the lambda expressions*

When we introduced lambda expressions in chapter 2, we presented them mainly as a new way to express anonymous delegates. We then demonstrated how they could be assigned to delegate types. Here is one more example:

```
Func<int, bool> isOdd = i => (i & 1) == 1;
```

Here we use the `Func<T, TResult>` generic delegate type defined in the `System` namespace. This type is declared as follows in the `System.Core.dll` assembly that comes with .NET 3.5:

```
delegate TResult Func<T, TResult>(T arg);
```

Our `isOdd` delegate object represents a method that takes an integer as a parameter and returns a Boolean. This delegate variable can be used like any other delegate:

```
for (int i = 0; i < 10; i++)
{
  if (isOdd(i))
    Console.WriteLine(i + " is odd");
  else
    Console.WriteLine(i + " is even");
}
```

One thing we'd like to stress at this point is that a lambda expression can also be used as *data* instead of code. This is what expression trees are about.

3.5.2 *What are expression trees?*

Consider the following line of code that uses the `Expression<TDelegate>` type defined in the `System.Linq.Expressions` namespace:

C#
```
Expression<Func<int, bool>> isOdd = i => (i & 1) == 1;
```

Here is the equivalent line of code in VB.NET:

VB.NET
```
Dim isOdd As Expression(Of Func(Of Integer, Boolean)) = _
  Function(i) (i And 1) = 1
```

This time, we can't use `isOdd` as a delegate. This is because it's not a delegate, but an *expression tree.*

It turns out that the compiler knows about this `Expression<TDelegate>` type and behaves differently than with delegate types such as `Func<T, TResult>`. Rather than compiling the lambda expression into IL code that evaluates the expression, it generates IL that constructs a tree of objects representing the expression.

Note that only lambda expressions with an expression body can be used as expression trees. Lambda expressions with a statement body are not convertible to expression trees. In the following example, the first lambda expression can be used to declare an expression tree because it has an expression body, whereas the second can't be used to declare an expression tree because it has a statement body (see chapter 2 for more details on the two kinds of lambda expressions):

```
Expression<Func<Object, Object>> identity = o => o;
Expression<Func<Object, Object>> identity = o => { return o; };
```

When the compiler sees a lambda expression being assigned to a variable of an `Expression<>` type, it will compile the lambda into a series of factory method calls that will build the expression tree at runtime. Here is the code that is generated behind the scenes by the compiler for our expression:

C#

```
ParameterExpression i = Expression.Parameter(typeof(int), "i");
Expression<Func<int, bool>> isOdd =
  Expression.Lambda<Func<int, bool>>(
    Expression.Equal(
      Expression.And(
        i,
        Expression.Constant(1, typeof(int))),
      Expression.Constant(1, typeof(int))),
    new ParameterExpression[] { i });
```

Here is the VB syntax:

VB.NET

```
Dim i As ParameterExpression = _
  Expression.Parameter(GetType(Integer), "i")
Dim isOdd As Expression(Of Func(Of Integer, Boolean)) = _
  Expression.Lambda(Of Func(Of Integer, Boolean))( _
    Expression.Equal( _
      Expression.And( _
        i, _
        Expression.Constant(1, GetType(Integer))), _
      Expression.Constant(1, GetType(Integer))), _
    New ParameterExpression() {i})
```

NOTE Expression trees are constructed at runtime when code like this executes, but once constructed they cannot be modified.

Note that you could write this code by yourself. It would be uninteresting for our example, but it could be useful for advanced scenarios. We'll keep that for chapter 5, where we use expression trees to create dynamic queries.

Apart from being grateful to the compiler for generating this for us, you can start to see why this is called an expression tree. Figure 3.5 is a graphical representation of this tree.

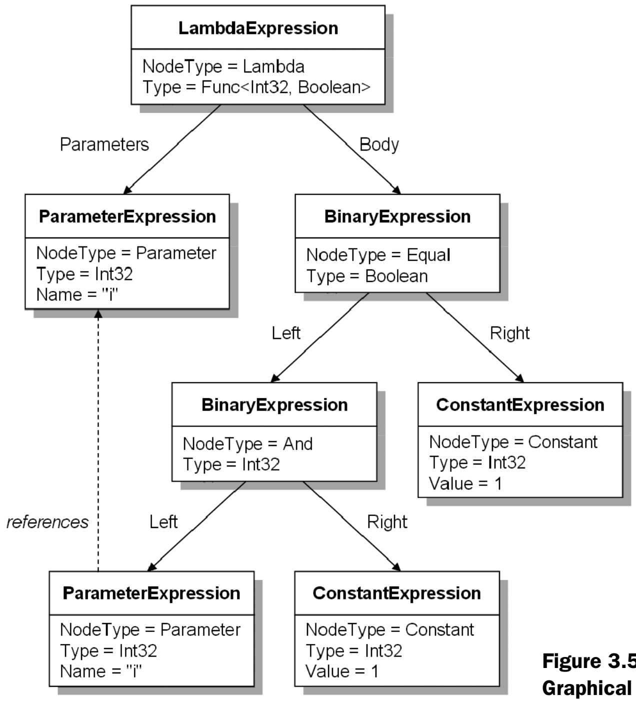


Figure 3.5
Graphical view of an expression tree

At this stage, you've learned that lambda expressions can be represented as code (delegates) or as data (expression trees). Assigned to a delegate, a lambda expression emits IL code; assigned to `Expression<TDelegate>`, it emits an expression tree, which is an in-memory data structure that represents the parsed lambda.

The best way to prove that an expression completely describes a lambda expression is to show how expression trees can be compiled down to delegates:

```
Func<int, bool> isOddDelegate = i => (i & 1) == 1;
Expression<Func<int, bool>> isOddExpression = i => (i & 1) == 1;
Func<int, bool> isOddCompiledExpression =
  isOddExpression.Compile();
```

In this code, `isOddDelegate` and `isOddCompiledExpression` are equivalent. Their IL code is the same.

The burning question at this point should be, "Why would we need expression trees?" Well, an expression is a kind of an *abstract syntax tree (AST)*. In computer science, an AST is a data structure that represents source code that has been parsed. An AST is often used as a compiler or interpreter's internal representation of a computer program while it is being optimized, from which code generation is performed. In our case, an expression tree is the result of the parsing operation the C# compiler does on a lambda expression. The goal here is that some code will analyze the expression tree to perform various operations.

Expression trees can be given to tools at runtime, which use them to guide their execution or translate them into something else, such as SQL in the case of LINQ to SQL. As you'll see in more detail in parts 4 and 5 of this book, LINQ to SQL uses information contained in expression trees to generate SQL and perform queries against a database.

For the moment, we'd like to point out that expression trees are another way to achieve deferred query execution.

3.5.3 IQueryable, deferred query execution redux

You've seen that one way to achieve deferred query execution is to rely on `IEnumerable<T>` and iterators. Expression trees are the basis for another way to out-of-process querying.

This is what is used in the case of LINQ to SQL. When we write code as follows, as we did in chapter 1, no SQL is executed before the `foreach` loop starts iterating on `contacts`:

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

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

foreach (var contact in contacts)
  Console.WriteLine("Bonjour "+contact.Name);
```

This behavior is similar to what happens with `IEnumerable<T>`, but this time, the type of `contacts` is not `IEnumerable<Contact>`, like you could expect, but `IQueryable<Contact>`. What happens with `IQueryable<T>` is different than with sequences. An instance of `IQueryable<T>` receives an expression tree it can inspect to decide what processing it should perform.

In this case, as soon as we start enumerating the content of `contacts`, the expression tree it contains gets analyzed, SQL is generated and executed, and the results of the database query are returned as `Contact` objects.

We won't go into detail about how things work here, but `IQueryable` is more powerful than sequences based on `IEnumerable` because intelligent processing based on the analysis of expression trees can happen. By examining a complete query through its expression tree representation, a tool can take smart decisions and make powerful optimizations. `IQueryable` and expression trees are suitable for cases where `IEnumerable` and its pipelining pattern are not flexible enough.

Deferred query execution with expression trees allow LINQ to SQL to optimize a query containing multiple nested or complex queries into the fewest number of efficient SQL statements possible. If LINQ to SQL were to use a pipelining pattern like the one supported by `IEnumerable<T>`, it would only be able to execute several small queries in cascade against databases instead of a reduced number of optimized queries.

As you'll see later, expression trees and `IQueryable` can be used to extend LINQ and are not limited to LINQ to SQL. We'll demonstrate how we can take advantage of LINQ's extensibility in chapter 12.

Now that we've explored all the main elements of LINQ, let's see where to find the nuts and bolts you need to build your applications.

3.6 LINQ DLLs and namespaces

The classes and interfaces that you need to use LINQ in your applications come distributed in a set of assemblies (DLLs) provided with .NET 3.5. You need to know what assemblies to reference and what namespaces to import.

The main assembly you'll use is `System.Core.dll`. In order to write LINQ to Objects queries, you'll need to import the `System.Linq` namespace it contains. This is how the standard query operators provided by the `System.Linq.Enumerable` class become available to your code. Note that the `System.Core.dll` assembly is referenced by default when you create a new project with Visual Studio 2008.

If you need to work with expression trees or create your own `IQueryable` implementation, you'll also need to import the `System.Linq.Expressions` namespace, which is also provided by the `System.Core.dll` assembly.

In order to work with LINQ to SQL or LINQ to XML, you have to use dedicated assemblies: respectively `System.Data.Linq.dll` or `System.Xml.Linq.dll`. LINQ's features for the `DataSet` class are provided by the `System.Data.DataSetExtensions.dll` assembly.

The `System.Xml.Linq.dll` and `System.Data.DataSetExtensions.dll` assemblies are referenced by default when you create projects with Visual Studio 2008. `System.Data.Linq.dll` is not referenced by default. You need to reference it manually.

Table 3.3 is an overview of the LINQ assemblies and namespaces, and their content.

Table 3.3 Content of the assemblies provided by .NET 3.5 that are useful for LINQ

File name	Namespaces	Description and content
System.Core.dll		
	`System`	`Action` and `Func` delegate types

	`System.Linq`	`Enumerable` class (extension methods for `IEnumerable<T>`)
		`IQueryable` and `IQueryable<T>` interfaces
		`Queryable` class (extension methods for `IQueryable<T>`)
		`IQueryProvider` interface
		`QueryExpression` class
		Companion interfaces and classes for query operators:
		`Grouping<TKey, TElement>`
		`ILookup<TKey, TElement>`
		`IOrderedEnumerable<TElement>`
		`IOrderedQueryable`
		`IOrderedQueryable<T>`
		`Lookup<TKey, TElement>`

	`System.Linq.Expressions`	`Expression<TDelegate>` class and other classes that enable expression trees
System.Data.DataSetExtensions.dll		
	`System.Data`	Classes for LINQ to DataSet, such as `TypedTableBase<T>`, `DataRowComparer`, `DataTableExtensions`, and `DataRowExtensions`
System.Data.Linq.dll		
	`System.Data.Linq`	Classes for LINQ to SQL, such as `DataContext`, `Table<TEntity>`, and `EntitySet<TEntity>`
	`System.Data.Linq.Mapping`	Classes and attributes for LINQ to SQL, such as `ColumnAttribute`, `FunctionAttribute`, and `TableAttribute`
	`System.Data.Linq.SqlClient`	The `SqlMethods` and `SqlHelpers` classes

Table 3.3 Content of the assemblies provided by .NET 3.5 that are useful for LINQ *(continued)*

File name	Namespaces	Description and content
System.Xml.Linq.dll		
	`System.Xml.Linq`	Classes for LINQ to XML, such as `XObject`, `XNode`, `XElement`, `XAttribute`, `XText`, `XDocument`, and `XStreamingElement`
	`System.Xml.Schema`	`Extensions` class that provides extension methods to deal with XML schemas
	`System.Xml.XPath`	`Extensions` class that provides extension methods to deal with XPath expressions and to create XPathNavigator objects from XNode instances

3.7 Summary

In this chapter, we've explained how LINQ extends C# and VB.NET, as well as the .NET Framework. You should now have a better idea of what LINQ is.

We've walked through some important foundational LINQ material. You've learned some new terminology and concepts.

Here is a summary of what we've introduced in this chapter:

- Sequences, which are enumerations and iterators applied to LINQ
- Deferred query execution
- Query operators, extension methods that allow operations in the context of LINQ queries
- Query expressions, which allow the SQL-like `from…where…select` syntax
- Expression trees, which represent queries as data and allow advanced extensibility

You're now prepared to read and write LINQ code. We'll now get to action and start using LINQ for useful things. In part 2, we'll use LINQ to Objects to query objects in memory. In part 3, we'll address persistence to relational databases with LINQ to SQL. In part 4, we'll detail how to work on XML documents with LINQ to XML.

MICROSOFT/.NET

LINQ in Action

Fabrice Marguerie • Steve Eichert • Jim Wooley

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

—FROM THE FOREWORD BY
Matt Warren
Principal Architect, Microsoft

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

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

What's Inside

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

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

"Great if you want to fully grok LINQ."

—Javier Lozano, lozanotek.com

"Very useful—both straightforward and pragmatic."

—Bruno Boucard
Microsoft France

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

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

"Covers LINQ, inside & out."

—Mohammad Azam
University of Houston

"A great guide to all things LINQ!"

—Tomas Restrepo, devdeo ltda

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

ISBN-13: 978-1933988160
ISBN-10: 1933988169
54499
9 781933 988160

MANNING $44.99 / Can $44.99