```
//
//
```

```
{ 1, 2, 3, 3, 4, 5 }
{ 1, 2, 3, 4, 5 }
```

Specifying the type argument explicitly is useful when the sequences are differently typed, but the elements have a common base type. For instance, with the reflection API (Chapter 18), methods and properties are represented with MethodInfo and PropertyInfo classes, which have a common base class called MemberInfo. We can concatenate methods and properties by stating that base class explicitly when calling Concat:

```
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();
IEnumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

In the next example, we filter the methods before concatenating:

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MemberInfo> (props);
```

Interestingly, this example compiles in C# 4.0 but not in C# 3.0 because it relies on interface type parameter variance: `methods` is of type `IEnumerable<MethodInfo>`, which requires a covariant conversion to `IEnumerable<MemberInfo>`. It's a good illustration of how variance makes things work more as you'd expect.

Intersect and Except

`Intersect` returns the elements that two sequences have in common. `Except` returns the elements in the first input sequence that are *not* present in the second:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
```

```
IEnumerable<int>
  commonality = seq1.Intersect (seq2),
  difference1 = seq1.Except    (seq2),
  difference2 = seq2.Except    (seq1);
```

```
//
//
//
```

```
{ 3 }
{ 1, 2 }
{ 4, 5 }
{ 4, 5 }
```

`Enumerable.Except` works internally by loading all of the elements in the first collection into a dictionary, then removing from the dictionary all elements present in the second sequence. The equivalent in SQL is a `NOT EXISTS` or `NOT IN` subquery:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

The Zip Operator

`IEnumerable<TFirst>`, `IEnumerable<TSecond>`→`IEnumerable<TResult>`

The `Zip` operator was added in Framework 4.0. It enumerates two sequences in step (like a zipper), returning a sequence based on applying a function over each element pair. For instance, the following:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

```
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

produces a sequence with the following elements:

```
3=three
5=five
7=seven
```

Extraneous elements in either input sequence are ignored. Zip is not supported when querying a database.

Conversion Methods

LINQ deals primarily in sequences—collections of type **IEnumerable<T>**, in other words. The conversion methods convert to and from other types of collections:

Method	Description
OfType	Converts IEnumerable to IEnumerable<T>, discarding wrongly typed elements
Cast	Converts IEnumerable to IEnumerable<T>, throwing an exception if there are any wrongly typed elements
ToArray	Converts IEnumerable<T> to T[]
ToList	Converts IEnumerable<T> to List<T>

ToDictionary

ToDictionary	Converts IEnumerable<T> to Dictionary<TKey,TValue>
ToLookup	Converts IEnumerable<T> to ILookup<TKey,TElement>
AsEnumerable	Downcasts to IEnumerable<T>
AsQueryable	Casts or converts to IQueryable<T>

OfType and Cast

OfType and Cast accept a nongeneric IEnumerable collection and emit a generic IEnumerable<T> sequence that you can subsequently query:

IEnumerable<T> sequence that you can subsequently query:

```
ArrayList classicList = new ArrayList();          // in System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Cast and OfType differ in their behavior when encountering an input element that's of an incompatible type. Cast throws an exception; OfType ignores the incompatible element. Continuing the preceding example:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
  sequence2 = classicList.OfType<int>(), // OK - ignores offending DateTime
  sequence3 = classicList.Cast<int>();   // Throws exception
```

The rules for element compatibility exactly follow those of C#'s is operator, and therefore consider only reference conversions and unboxing conversions. We can see this by examining the internal implementation of OfType:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

```
        yield return (TSource)element;
    }
```

Cast has an identical implementation, except that it omits the type compatibility test:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

A consequence of these implementations is that you cannot use Cast to perform numeric or custom conversions (for these, you must perform a Select operation instead). In other words, Cast is not as flexible as C#'s cast operator:

```
int i = 3;
long l = i;
int i2 = (int) l;
int i2 = (int) l;
```

```
// Implicit numeric conversion int->long
// Explicit numeric conversion long->int
```

We can demonstrate this by attempting to use OfType or Cast to convert a sequence of ints to a sequence of longs:

```
    int[] integers = { 1, 2, 3 };
```

```
IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

When enumerated, test1 emits zero elements and test2 throws an exception. Examining OfType's implementation, it's fairly clear why. After substituting TSource, we get the following expression:

```
    (element is long)
```

which returns false for an int element, due to the lack of an inheritance relationship.

which returns `false` for an `int element`, due to the lack of an inheritance relationship.

The reason for `test2` throwing an exception, when enumerated, is subtler. Notice in `Cast`'s implementation that `element` is of type `object`. When `TSource` is a value type, the CLR assumes this is an *unboxing conversion*, and synthesizes a method that reproduces the scenario described in the section "Boxing and Unboxing" on page 86 in Chapter 3:

```
int value = 123;
object element = value;
long result = (long) element;
// exception
```

Because the `element` variable is declared of type `object`, an `object`-to-`long` cast is performed (an unboxing) rather than an object-to-long cast is performed (an unboxing) rather than an int-to-long numeric conversion. Unboxing operations require an exact type match, so the object-to-long unbox fails when given an int.

As we suggested previously, the solution is to use an ordinary Select:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

OfType and Cast are also useful in downcasting elements in a generic input sequence. For instance, if you have an input sequence of type IEnumerable<Fruit>, OfType<Apple> would return just the apples. This is particularly useful in LINQ to XML (see Chapter 10).

Cast has query syntax support: simply precede the range variable with a type:

```
from TreeNode node in myTreeView.Nodes
...
```

ToArray, ToList, ToDictionary, and ToLookup

ToArray and ToList emit the results into an array or generic list. These operators force the immediate enumeration of the input sequence (unless indirected via a subquery or expression tree). For examples, refer to the section "Deferred Execuquery or expression tree). For examples, refer to the section "Deferred Execution" on page 324 in Chapter 8.

ToDictionary and ToLookup accept the following arguments:

Argument	Type
Input sequence	IEnumerable<TSource>
Key selector	TSource => TKey
Element selector (optional)	TSource => TElement
Comparer (optional)	IEqualityComparer<TKey>

ToDictionary also forces immediate execution of a sequence, writing the results to a generic Dictionary. The keySelector expression you provide must evaluate to a unique value for each element in the input sequence; otherwise, an exception is thrown. In contrast, ToLookup allows many elements of the same key. We describe lookups in the earlier section "Joining with lookups" on page 392.

lookups in the earlier section "Joining with lookups" on page 392.

AsEnumerable and AsQueryable

AsEnumerable upcasts a sequence to IEnumerable<T>, forcing the compiler to bind subsequent query operators to methods in Enumerable, instead of Queryable. For an example, see the section "Combining Interpreted and Local Queries" on page 343 in Chapter 8.

AsQueryable downcasts a sequence to IQueryable<T> if it implements that interface. Otherwise, it instantiates an IQueryable<T> wrapper over the local query.

Element Operators

IEnumerable<TSource>→TSource

Method	Description	SQL equivalents
First, FirstOrDefault	Returns the first element in the sequence, optionally sat-	SELECT TOP 1 ... ORDER BY ...

First, FirstOrDefault	Returns the first element in the sequence, optionally satisfying a predicate	SELECT TOP 1...ORDER BY...
Last, LastOrDefault	Returns the last element in the sequence, optionally satisfying a predicate	SELECT TOP 1...ORDER BY...DESC

Method	Description	SQL equivalents
Single, SingleOrDefault	Equivalent to First/ FirstOrDefault, but throws an exception if there is more than one match	
ElementAt, ElementAtOrDefault	Returns the element at the specified position	Exception thrown
DefaultIfEmpty	Returns null or default(TSource) if the sequence has no elements	OUTER JOIN

Methods ending in "OrDefault" return default(TSource) rather than throwing an

Methods ending in “OrDefault” return `default(TSource)` rather than throwing an exception if the input sequence is empty or if no elements match the supplied predicate.

`default(TSource)`is `null` for reference type elements, or “blank” (usually zero) for value type elements.

First, Last, and Single

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

The following example demonstrates `First` and `Last`:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int first      = numbers.First();
```

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int first      = numbers.First();
int last       = numbers.Last();
int firstEven  = numbers.First  (n => n % 2 == 0);
int lastEven   = numbers.Last   (n => n % 2 == 0);
```

```
// 1
// 5
// 2
// 4
```

The following demonstrates `First` versus `FirstOrDefault`:

```
int firstBigError  = numbers.First          (n => n > 10);
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);
```

```
// Exception
// Exception
// 0
```

To avoid an exception, Single requires exactly one matching element; SingleOrDefault requires one *or zero* matching elements:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0);   // 3
int divBy2Err  = numbers.Single (n => n % 2 == 0);   // Error: 2 & 4 match
```

```
int singleError = numbers.Single          (n => n > 10);
int noMatches   = numbers.SingleOrDefault (n => n > 10);
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);
```

```
// Error
// 0
// Error
```

Single is the "fussiest" in this family of element operators. FirstOrDefault and LastOrDefault are the most tolerant.

Single is the “fussiest” in this family of element operators. FirstOrDefault and LastOrDefault are the most tolerant.

In LINQ to SQL and EF, Single is often used to retrieve a row from a table by primary key:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

ElementAt

Argument	Type
Source sequence	IEnumerable<TSource>
Index of element to return	int

ElementAt picks the *n*th element from the sequence:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);
int tenthError = numbers.ElementAt (9);
int tenth      = numbers.ElementAtOrDefault (9);
```

```
// 3
// Exception
// Exception
// 0
```

`Enumerable.ElementAt` is written such that if the input sequence happens to implement `IList<T>`, it calls `IList<T>`’s indexer. Otherwise, it enumerates *n* times, and then returns the next element. `ElementAt` is not supported in LINQ to SQL or EF.

DefaultIfEmpty

`DefaultIfEmpty` converts empty sequences to null/`default()`. This is used in writing flat outer joins: see the earlier sections “Outer joins with SelectMany” on page 385 and “Flat outer joins” on page 392.

Aggregation Methods

`IEnumerable<TSource>`→ *`scalar`*

IEnumerable<TSource> → scalar

Method	Description	SQL equivalents
`Count, LongCount`	Returns the number of elements in the input sequence, optionally satisfying a predicate	`COUNT (...)`
`Min, Max`	Returns the smallest or largest element in the sequence	`MIN (...), MAX (...)`
`Sum, Average`	Calculates a numeric sum or average over elements in the sequence	`SUM (...), AVG (...)`
`Aggregate`	Performs a custom aggregation	Exception thrown

Count and LongCount

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

Count simply enumerates over a sequence, returning the number of items:

```
int fullCount = new int[] { 5, 6, 7 }.Count();    // 3
```

The internal implementation of Enumerable.Count tests the input sequence to see whether it happens to implement ICollection<T>. If it does, it simply calls ICollection<T>.Count. Otherwise, it enumerates over every item, incrementing a counter.

You can optionally supply a predicate:

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c));
```

```
// 3
```

LongCount does the same job as Count, but returns a 64-bit integer, allowing for sequences of greater than 2 billion elements.

Min and Max

Min and Max

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector (optional)	`TSource => TResult`

`Min` and `Max` return the smallest or largest element from a sequence:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min();  // 14;
int largest  = numbers.Max();  // 32;
```

If you include a `selector` expression, each element is first projected:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

A `selector` expression is mandatory if the items themselves are not intrinsically comparable—in other words, if they do not implement `IComparable<T>`:

```
Purchase runtimeError = dataContext.Purchases.Min ();           // Error
```

comparable—in other words, if they do not implement IComparable<T>.

```
Purchase runtimeError = dataContext.Purchases.Min ();            // Error
decimal? lowestPrice = dataContext.Purchases.Min (p => p.Price);  // OK
```

A selector expression determines not only how elements are compared, but also the final result. In the preceding example, the final result is a decimal value, not a purchase object. To get the cheapest purchase, you need a subquery:

```
Purchase cheapest = dataContext.Purchases
  .Where (p => p.Price == dataContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

In this case, you could also formulate the query without an aggregation—using an OrderBy followed by FirstOrDefault.

Sum and Average

Argument	Type
Source sequence	IEnumerable<TSource>

Source sequence	`IEnumerable<TSource>`
Result selector (optional)	`TSource => TResult`

Sum and Average are aggregation operators that are used in a similar manner to Min and Max:

and Max:

```
decimal[] numbers
decimal sumTotal
decimal average
```

```
= { 3, 4, 8 };
= numbers.Sum();
= numbers.Average();
```

```
// 15
// 5
```

```
(mean value)
```

(mean value)

The following returns the total length of each of the strings in the names array:

```
int combinedLength = names.Sum (s => s.Length);   // 19
```

Sum and Average are fairly restrictive in their typing. Their definitions are hard-wired to each of the numeric types (int, long, float, double, decimal, and their nullable versions). In contrast, Min and Max can operate directly on anything that implements IComparable<T>—such as a string, for instance.

Further, Average always returns either decimal or double, according to the following table:

Selector type	Result type
decimal	decimal
int, long, float, double	double

This means the following does not compile (“cannot convert double to int”):

This means the following does not compile ("cannot convert double to int"):

```
int avg = new int[] { 3, 4 }.Average();
```

But this will compile:

```
double avg = new int[] { 3, 4 }.Average();
```

```
// 3.5
```

Average implicitly upscales the input values to avoid loss of precision. In this example, we averaged integers and got 3.5, without needing to resort to an input element cast:

```
double avg = numbers.Average (n => (double) n);
```

When querying a database, Sum and Average translate to the standard SQL aggregations. The following query returns customers whose average purchase was more than $500:

than $500:

```
from c in dataContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

Aggregate

Aggregate allows you to specify a custom accumulation algorithm for implementing unusual aggregations. Aggregate is not supported in LINQ to SQL or Entity Framework, and is somewhat specialized in its use cases. The following demonstrates how Aggregate can do the work of Sum:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 9
```

The first argument to Aggregate is the *seed*, from which accumulation starts. The second argument is an expression to update the accumulated value, given a fresh element. You can optionally supply a third argument to project the final result value second argument is an expression to update the accumulated value, given a fresh element. You can optionally supply a third argument to project the final result value from the accumulated value.

Most problems for which Aggregate has been designed can be solved as easily with a foreach loop—and with more familiar syntax. The advantage of using Aggregate is that with large or complex aggregations, you can automatically parallelize the operation with PLINQ (see Chapter 22).

Unseeded aggregations

You can omit the seed value when calling Aggregate, in which case the first element becomes the *implicit* seed, and aggregation proceeds from the second element. Here's the preceding example, *unseeded*:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);   // 6
```

This gives the same result as before, but we're actually doing a *different calculation*. Before, we were calculating 0+1+2+3; now we're calculating 1+2+3. We can better illustrate the difference by multiplying instead of adding:

better illustrate the difference by multiplying instead of adding:

```
int[] numbers = { 1, 2, 3 };
 int x = numbers.Aggregate (0, (prod, n) => prod * n);   // 0*1*2*3 = 0
 int y = numbers.Aggregate (   (prod, n) => prod * n);   //   1*2*3 = 6
```

As we'll see in Chapter 22, unseeded aggregations have the advantage of being parallelizable without requiring the use of special overloads. However, there are some traps with unseeded aggregations.

Traps with unseeded aggregations

The unseeded aggregation methods are intended for use with delegates that are *commutative* and *associative*. If used otherwise, the result is either *unintuitive* (with ordinary queries) or *nondeterministic* (in the case that you parallelize the query with PLINQ). For example, consider the following function:

```
(total, n) => total + n * n
```

This is neither commutative nor associative. (For example, 1+2*2 != 2+1*1). Let's see what happens when we use it to sum the square of the numbers 2, 3, and 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);      // 27
```

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);    // 27
```

Instead of calculating:

```
2*2 + 3*3 + 4*4

// 29
// 27
```

it calculates:

```
2 + 3*3 + 4*4
```

We can fix this in a number of ways. First, we could include 0 as the first element:

```
int[] numbers = { 0, 2, 3, 4 };
```

Not only is this inelegant, but it will still give incorrect results if parallelized—because PLINQ leverages the function's assumed associativity by selecting *multiple* elements as seeds. To illustrate, if we denote our aggregation function as follows:

```
f(total, n) => total + n * n
```

```
f(total, n) => total + n * n
```

then LINQ to Objects would calculate this:

```
f(f(f(0, 2),3),4)
```

whereas PLINQ may do this:

```
f(f(0,2),f(3,4))
```

with the following result:

```
First partition:
Second partition:
Final result:
OR EVEN:
```

```
a = 0 + 2*2
b = 3 + 4*4
        a + b*b
        b + a*a
```

```
(= 4)
(= 19)
(= 365!)
(= 35!)
```

There are two good solutions. The first is to turn this into a seeded aggregation—with zero as the seed. The only complication is that with PLINQ, we'd need to use a special overload in order for the query not to execute sequentially (see "Optimizing PLINQ" on page 886 in Chapter 22).

PLINQ” on page 886 in Chapter 22).

The second solution is to restructure the query such that the aggregation function is commutative and associative:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```

Of course, in such simple scenarios you can (and should) use the Sum operator instead of Aggregate:

```
int sum = numbers.Sum (n => n * n);
```

You can actually go quite far just with Sum and Average. For instance, you can use Average to calculate a root-mean-square:

```
Math.Sqrt (numbers.Average (n => n * n))
```

and even standard deviation:

```
double mean = numbers.Average();
double mean = numbers.Average();
double sdev = Math.Sqrt (numbers.Average (n =>
              {
                double dif = n - mean;
                return dif * dif;
              }));
```

Both are safe, efficient and fully parallelizable. In Chapter 22, we'll give a practical example of a custom aggregation that can't be reduced to Sum or Average.

Quantifiers

IEnumerable<TSource>→*bool*

IEnumerable<TSource>→*bool*

Method	Description	SQL equivalents
Contains	Returns true if the input sequence contains the given element	WHERE ... IN (...)
Any	Returns true if any elements satisfy the given predicate	WHERE ... IN (...)
All	Returns true if all elements satisfy the given predicate	WHERE (...)
SequenceEqual	Returns true if the second sequence has identical elements to the input sequence	

Contains and Any

The Contains method accepts an argument of type TSource; Any accepts an optional *predicate*.

Contains returns true if the given element is present:

```
    bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);
// true;
// true;
```

Any returns true if the given expression is true for at least one element. We can rewrite the preceding query with Any as follows:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);  // true;
```

Any can do everything that Contains can do, and more:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);
```

```
// false;
```

Calling Any without a predicate returns true if the sequence has one or more elements. Here's another way to write the preceding query:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

Any is particularly useful in subqueries and is used often when querying databases, for example:

```
from c in dataContext.Customers
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

All and SequenceEqual

All returns true if all elements satisfy a predicate. The following returns customers whose purchases are less than $100:

```
    dataContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

SequenceEqual compares two sequences. To return true, each sequence must have identical elements, in the identical order.

Generation Methods

`void→IEnumerable<TResult>`

Method	Description
Empty	Creates an empty sequence
Repeat	Creates a sequence of repeating elements

Repeat	Creates a sequence of repeating elements
Range	Creates a sequence of integers

Empty, Repeat, and Range are static (nonextension) methods that manufacture simple local sequences.

Empty

Empty manufactures an empty sequence and requires just a type argument:

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);                              // <nothing>
```

In conjunction with the ?? operator, Empty does the reverse of DefaultIfEmpty. For example, suppose we have a jagged array of integers, and we want to get all the integers into a single flat list. The following SelectMany query fails if any of the inner arrays is null:

```
int[][] numbers =
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null                     // this null makes the query below fail.
};
```

```
IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Empty in conjunction with ?? fixes the problem:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());
```

```
foreach (int i in flat)
  Console.Write (i + " ");
```

```
// 1 2 3 4 5 6
```

Range and Repeat

Range and Repeat

Range and Repeat work only with integers. Range accepts a starting index and count:

```
foreach (int i in Enumerable.Range (5, 5))
  Console.Write (i + " ");                          // 5 6 7 8 9
```

Repeat accepts the number to repeat, and the number of iterations:

```
foreach (int i in Enumerable.Repeat (5, 3))
  Console.Write (i + " ");                          // 5 5 5
```

1

10

LINQ to XML

The .NET Framework provides a number of APIs for working with XML data. From Framework 3.5, the primary choice for general-purpose XML document processing is *LINQ to XML*. LINQ to XML comprises a lightweight LINQ-friendly XML docu-

Framework 3.5, the primary choice for general-purpose XML document processing is *LINQ to XML*. LINQ to XML comprises a lightweight LINQ-friendly XML document object model, and a set of supplementary query operators. In most scenarios, it can be considered a complete replacement for the preceding W3C-compliant DOM, a.k.a. `XmlDocument`.

In this chapter, we concentrate entirely on LINQ to XML. In the following chapter, we cover the more specialized XML types and APIs, including the forward-only reader/writer, the types for working with schemas, stylesheets and XPaths, and the legacy W3C-compliant DOM.

The LINQ to XML DOM is extremely well designed and highly performant. Even without LINQ, the LINQ to XML DOM is valuable as a lightweight façade over the low-level `XmlReader` and `XmlWriter` classes.

All LINQ to XML types are defined in the `System.Xml.Linq` namespace.

Architectural Overview

Architectural Overview

This section starts with a very brief introduction to the concept of a DOM, and then explains the rationale behind LINQ to XML's DOM.

What Is a DOM?

Consider the following XML file:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

As with all XML files, we start with a *declaration*, and then a root *element*, whose name is customer. It has two *attributes*, each with a name (id and status) and value

As with all XML files, we start with a *declaration*, and then a root *element*, whose name is `customer`. It has two *attributes*, each with a name (`id` and `status`) and value (`"123"` and `"archived"`). Within `customer`, there are two child elements, `firstname` and `lastname`, each having simple text content (`"Joe"` and `"Bloggs"`).

Each of these constructs—declaration, element, attribute, value, and text content—can be represented with a class. And if such classes have collection properties for storing child content, we can assemble a *tree* of objects to fully describe a document. This is called a *document object model*, or DOM.

The LINQ to XML DOM

LINQ to XML comprises two things:

•

•

An XML DOM, which we call the *X-DOM*

A set of about 10 supplementary query operators

A set of about 10 supplementary query operators

As you might expect, the X-DOM consists of types such as `XDocument`, `XElement`, and `XAttribute`. Interestingly, the X-DOM types are not tied to LINQ—you can load, instantiate, update, and save an X-DOM without ever writing a LINQ query.

Conversely, you could use LINQ to query a DOM created of the older W3C-compliant types. However, this would be frustrating and limiting. The distinguishing feature of the X-DOM is that it's *LINQ-friendly*. This means:

- It has methods that emit useful `IEnumerable` sequences, upon which you can query.
- Its constructors are designed such that you can build an X-DOM tree through a LINQ projection.

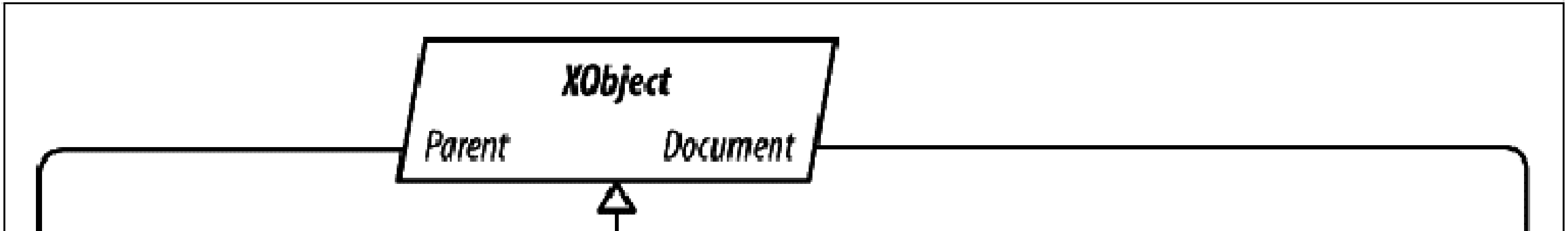
X-DOM Overview

X-DOM Overview

Figure 10-1 shows the core X-DOM types. The most frequently used of these types is XElement. XObject is the root of the *inheritance* hierarchy; XElement and XDocument are roots of the *containership* hierarchy. Figure 10-2 shows the X-DOM tree created from the following code:

```
string xml = @"<customer id='123' status='archived'>
                 <firstname>Joe</firstname>
                 <lastname>Bloggs<!--nice name--></lastname>
               </customer>";
```

```
XElement customer = XElement.Parse (xml);
```



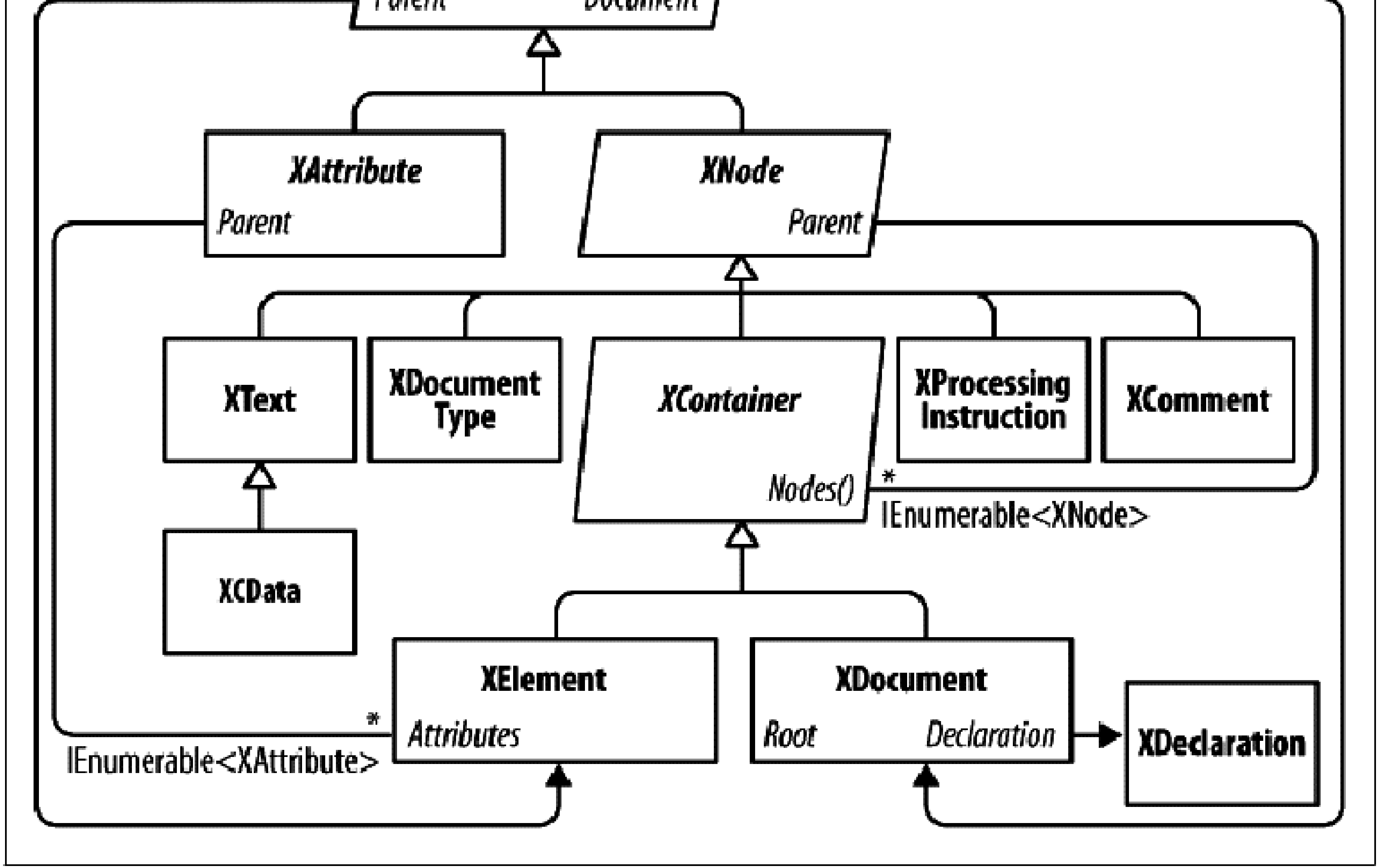
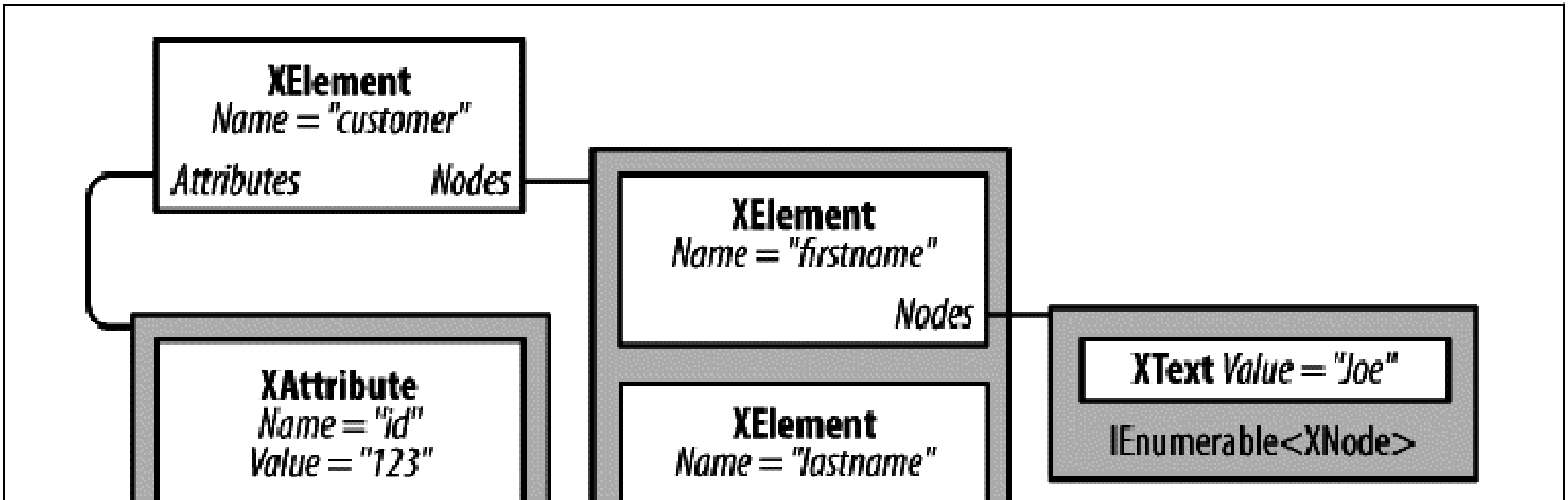


Figure 10-1. Core X-DOM types

[redacted]



XElement
Name = "customer"
Attributes
Nodes
XElement
Name = "firstname"
Nodes
XText Value = "Joe"
IEnumerable<XNode>
XAttribute
Name = "id"
Value = "123"
XElement
Name = "lastname"

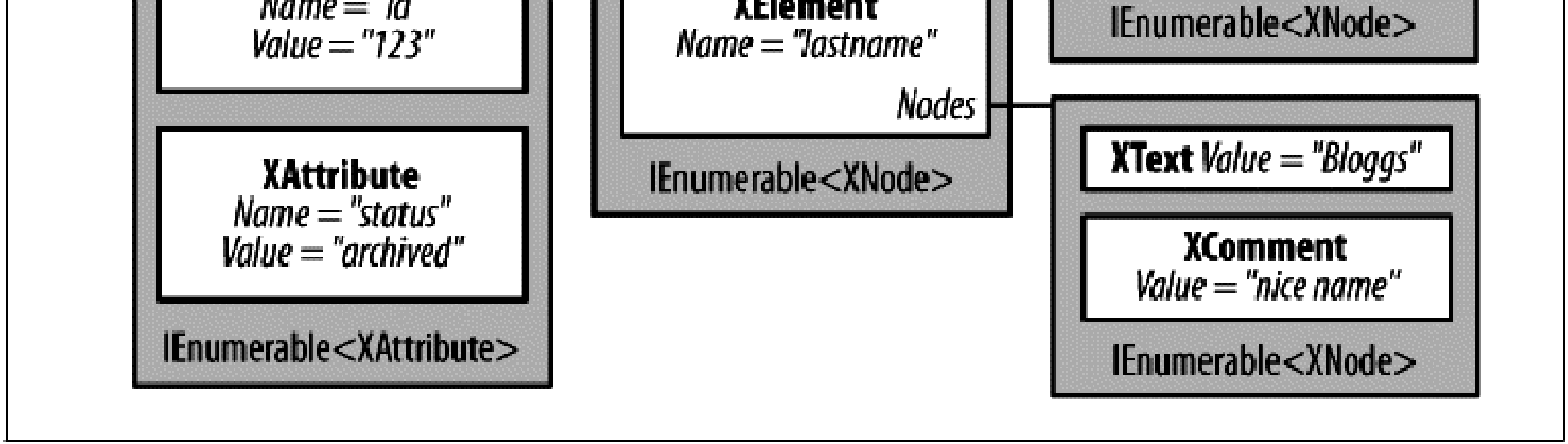


Figure 10-2. A simple X-DOM tree

XObject is the abstract base class for all XML content. It defines a link to the Parent element in the containership tree as well as an optional XDocument.

XNode is the base class for most XML content excluding attributes. The distinguishing feature of XNode is that it can sit in an ordered collection of mixed-type XNodes. For instance, consider the following XML:

```
<data>
  Hello world
```

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

Within the parent element `<data>`, there's first an `XText` node (`Hello world`), then an `XElement` node, then an `XComment` node, and then a second `XElement` node. In contrast, an `XAttribute` will tolerate only other `XAttributes` as peers.

Although an `XNode` can access its parent `XElement`, it has no concept of *child* nodes: this is the job of its subclass `XContainer`. `XContainer` defines members for dealing with children and is the abstract base class for `XElement` and `XDocument`.

`XElement` introduces members for managing attributes—as well as a `Name` and `Value`. In the (fairly common) case of an element having a single `XText` child node, the `Value` property on `XElement` encapsulates this child's content for both get and set

Value. In the (fairly common) case of an element having a single XText child node, the Value property on XElement encapsulates this child's content for both get and set operations, cutting unnecessary navigation. Thanks to Value, you can mostly avoid working directly with XText nodes.

XDocument represents the root of an XML tree. More precisely, it *wraps* the root XElement, adding an XDeclaration, processing instructions, and other root-level "fluff." Unlike with the W3C DOM, its use is optional: you can load, manipulate, and save an X-DOM without ever creating an XDocument! The nonreliance on XDocument also means you can efficiently and easily move a node subtree to another X-DOM hierarchy.

Loading and Parsing

Both XElement and XDocument provide static Load and Parse methods to build an X-DOM tree from an existing source:

-
-

Load builds an X-DOM from a file, URI, Stream, TextReader, or XmlReader.

Load builds an X-DOM from a file, URI, Stream, TextReader, or XmlReader.

Parse builds an X-DOM from a string.

For example:

```
XDocument fromWeb = XDocument.Load ("http://albahari.com/sample.xml");
```

```
XElement fromFile = XElement.Load (@"e:\media\somefile.xml");
```

```
XElement config = XElement.Parse (
@"<configuration>
    <client enabled='true'>
      <timeout>30</timeout>
    </client>
  </configuration>");
```

In later sections, we describe how to traverse and update an X-DOM. As a quick preview, here's how to manipulate the config element we just populated:

```
foreach (XElement child in config.Elements())
  Console.WriteLine (child.Name);                  // client
```

```
XElement client = config.Element ("client");

bool enabled = (bool) client.Attribute ("enabled");
Console.WriteLine (enabled);
client.Attribute ("enabled").SetValue (!enabled);

int timeout = (int) client.Element ("timeout");
Console.WriteLine (timeout);
client.Element ("timeout").SetValue (timeout * 2);

client.Add (new XElement ("retries", 3));
```

```
// Read attribute
// True
// Update attribute
// Read element
// 30
// Update element
// Add new element
```

```
Console.WriteLine (config);
// Implicitly call config.ToString()
```

Here’s the result of that last Console.WriteLine:

```
<configuration>
  <client enabled="false">
    <timeout>60</timeout>
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration
```

XNode also provides a static ReadFrom method that instantiates and populates any type of node from an XmlReader. Unlike Load, it stops after reading one (complete) node, so you can continue to read manually from the XmlReader afterward.

You can also do the reverse and use an XmlReader or XmlWriter to read or write an XNode, via its CreateReader and CreateWriter methods.

We describe XML readers and writers and how to use them with the X-DOM in Chapter 11

We describe XML readers and writers and how to use them with the X-DOM in Chapter 11.

Saving and Serializing

Calling ToString on any node converts its content to an XML string—formatted with line breaks and indentation as we just saw. (You can disable the line breaks and indentation by specifying SaveOptions.DisableFormatting when calling ToString.)

XElement and XDocument also provide a Save method that writes an X-DOM to a file, Stream, TextWriter, or XmlWriter. If you specify a file, an XML declaration is automatically written. There is also a WriteTo method defined in the XNode class, which accepts just an XmlWriter.

We describe the handling of XML declarations when saving in more detail in the section "Documents and Declarations" on page 431 later in this chapter.

Instantiating an X-DOM

Instantiating an X-DOM

Rather than using the Load or Parse methods, you can build an X-DOM tree by manually instantiating objects and adding them to a parent via XContainer's Add method.

To construct an XElement and XAttribute, simply provide a name and value:

```
XElement lastName = new XElement ("lastname", "Bloggs");
lastName.Add (new XComment ("nice name"));
```

```
XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
customer.Add (new XElement ("firstname", "Joe"));
customer.Add (lastName);

Console.WriteLine (customer.ToString());
```

The result:

The result:

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

A value is optional when constructing an `XElement`—you can provide just the element name and add content later. Notice that when we did provide a value, a simple string sufficed—we didn't need to explicitly create and add an `XText` child node. The X-DOM does this work automatically, so you can deal simply with "values."

Functional Construction

In our preceding example, it's hard to glean the XML structure from the code. X-DOM supports another mode of instantiation, called *functional construction* (from functional programming). With functional construction, you build an entire tree in a single expression:

functional programming). With functional construction, you build an entire tree in a single expression:

```
XElement customer =
  new XElement ("customer", new XAttribute ("id", 123),
    new XElement ("firstname", "joe"),
    new XElement ("lastname", "bloggs",
      new XComment ("nice name")
    )
  );
```

This has two benefits. First, the code resembles the shape of the XML. Second, it can be incorporated into the select clause of a LINQ query. For example, the following LINQ to SQL query projects directly into an X-DOM:

```
XElement query =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("firstname", c.FirstName)
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("firstname", c.FirstName),
```

```
);
```

```
  new XElement ("lastname", c.LastName,
    new XComment ("nice name")
  )
)
```

More on this later in this DOM” on page 441.

chapter

chapter,

in

the

section

“Projecting into an X-

LINQ to X

Specifying Content

Functional construction is possible because the constructors for `XElement` (and `XDocument`) are overloaded to accept a `params` object array:

```
public XElement (XName name, params object[] content)
```

The same holds true for the `Add` method in `XContainer`:

```
public void Add (params object[] content)
```

Hence, you can specify any number of child objects of any type when building or appending an X-DOM. This works because *anything* counts as legal content. To see

Hence, you can specify any number of child objects of any type when building or appending an X-DOM. This works because *anything* counts as legal content. To see how, we need to examine how each content object is processed internally. Here are the decisions made by XContainer, in order:

1. If the object is null, it's ignored.
2. If the object is based on XNode or XStreamingElement, it's added as is to the Nodes collection.
3. If the object is an XAttribute, it's added to the Attributes collection.
4. If the object is a string, it gets wrapped in an XText node and added to Nodes.[*]
5. If the object implements IEnumerable, it's enumerated, and the same rules are applied to each element.
6. Otherwise, the object is converted to a string, wrapped in an XText node, and then added to Nodes.[†]

Everything ends up in one of two buckets: Nodes or Attributes. Furthermore, any object is valid content because it can always ultimately call ToString on it and treat it as an XText node.

[*] The X-DOM actually optimizes this step internally by storing simple text content in a string. The XTEXT node is not actually created until you call Nodes( ) on the XContainer.

[*] The X-DOM actually optimizes this step internally by storing simple text content in a string. The XTEXT node is not actually created until you call Nodes( ) on the XContainer.

[†] See the previous footnote.

Before calling ToString on an arbitrary type, XContainer first tests whether it is one of the following types:

```
float, double, decimal, bool,
DateTime, DateTimeOffset, TimeSpan
```

If so, it calls an appropriate typed ToString method on the XmlConvert helper class instead of calling ToString on the object itself. This ensures that the data is round-trippable and compliant with standard XML formatting rules.

Automatic Deep Cloning

Automatic Deep Cloning

When a node or attribute is added to an element (whether via functional construction or an `Add` method), the node or attribute's `Parent` property is set to that element. A node can have only one parent element: if you add an already parented node to a second parent, the node is automatically *deep-cloned*. In the following example, each customer has a separate copy of `address`:

```
var address = new XElement ("address",
                  new XElement ("street", "Lawley St"),
                  new XElement ("town", "North Beach")
              );
var customer1 = new XElement ("customer1", address);
var customer2 = new XElement ("customer2", address);
```

```
customer1.Element ("address").Element ("street").Value = "Another St";
Console.WriteLine (
  customer2.Element ("address").Element ("street").Value);   // Lawley St
```

This automatic duplication keeps X-DOM object instantiation free of side effects—another hallmark of functional programming.

Navigating and Querying

As you might expect, the `XNode` and `XContainer` classes define methods and properties for traversing the X-DOM tree. Unlike a conventional DOM, however, these functions don't return a collection that implements `IList<T>`. Instead, they return either a single value or a *sequence* that implements `IEnumerable<T>`—upon which you are then expected to execute a LINQ query (or enumerate with a `foreach`). This allows for advanced queries as well as simple navigation tasks—using familiar LINQ query syntax.

Element and attribute names are case-sensitive in the X-DOM—just as they are in XML.

Child Node Navigation

Child Node Navigation

Return type	Members	Works on
XNode	FirstNode { get; }	XContainer
	LastNode { get; }	XContainer
IEnumerable<XNode>	Nodes()	XContainer*

XElement
IEnumerable<XElement>

DescendantNodes()

DescendantNodesAndSelf()

Element (XName)

XContainer*
XElement*
XContainer
XContainer*

XContainer*

XContainer*

XContainer*

XContainer*

Elements()

Elements (XName)

Descendants()

Descendants (XName)

	DescendantsAndSelf()	XElement*
	DescendantsAndSelf (XName)	XElement*
bool	HasElements { get; }	XElement

bool	HasElements { get; }	XElement

Functions marked with an asterisk in the third column of this and other tables also operate on *sequences* of the same type. For instance, you can call Nodes on either an XContainer or a sequence of XContainer objects. This is possible because of extension methods defined in System.Xml.Linq—the supplementary query operators we talked about in the overview.

FirstNode, LastNode, and Nodes

FirstNode and LastNode give you direct access to the first or last child node; Nodes returns all children as a sequence. All three functions consider only direct descendants. For example:

```
var bench = new XElement ("bench",
              new XElement ("toolbox",
                new XElement ("handtool", "Hammer"),
                new XElement ("handtool", "Rasp")
              ),
              new XElement ("toolbox",
                new XElement ("handtool", "Saw"),
              ),
              new XElement ("toolbox",
                new XElement ("handtool", "Saw"),
                new XElement ("powertool", "Nailgun")
              ),
              new XComment ("Be careful with the nailgun")
            );
foreach (XNode node in bench.Nodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting) + ".");
```

This is the output:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>.
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>.
<!--Be careful with the nailgun-->.
```

Retrieving elements

The Elements method returns just the child nodes of type XElement:

```
foreach (XElement e in bench.Elements())
  Console.WriteLine (e.Name + "=" + e.Value);    // toolbox=HammerRasp
```

```
foreach (XElement e in bench.Elements())
  Console.WriteLine (e.Name + "=" + e.Value);    // toolbox=HammerRasp
                                                 // toolbox=SawNailgun
```

The following LINQ query finds the toolbox with the nail gun:

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  where toolbox.Elements().Any (tool => tool.Value == "Nailgun")
  select toolbox.Value;
```

```
RESULT: { "SawNailgun" }
```

The next example uses a SelectMany query to retrieve the hand tools in all toolboxes:

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  from tool in toolbox.Elements()
  where tool.Name == "handtool"
```

```
from tool in toolbox.Elements()
where tool.Name == "handtool"
select tool.Value;
```

```
RESULT: { "Hammer", "Rasp", "Saw" }
```

Elements itself is equivalent to a LINQ query on Nodes. Our preceding query could be started as follows:

```
from toolbox in bench.Nodes().OfType<XElement>()
where ...
```

Elements can also return just the elements of a given name. For example:

```
int x = bench.Elements ("toolbox").Count();    // 2
```

This is equivalent to:

```
int x = bench.Elements().Where (e => e.Name == "toolbox").Count();
```

```
// 2
```

Elements is also defined as an extension method accepting IEnumerable<XContainer> or, more precisely, it accepts an argument of this type:

```
IEnumerable<T> where T : XContainer
```

This allows it to work with sequences of elements, too. Using this method, we can rewrite the query that finds the hand tools in all toolboxes as follows:

```
from tool in bench.Elements ("toolbox").Elements ("handtool")
select tool.Value.ToUpper();
```

The first call to Elements binds to XContainer's instance method; the second call to Elements binds to the extension method.

Retrieving a single element

The method Element (singular) returns the first matching element of the given name.

The method Element (singular) returns the first matching element of the given name. Element is useful for simple navigation, as follows:

```
XElement settings = XElement.Load ("databaseSettings.xml");
string cx = settings.Element ("database").Element ("connectString").Value;
```

Element is equivalent to calling Elements() and then applying FirstOrDefault query operator with a name-matching predicate. returns null if the requested element doesn't exist.

FirstOrDefault query operator with a name-matching predicate. returns `null` if the requested element doesn’t exist.

LINQ’s

`Element`

`Element("xyz").Value` will throw a `NullReferenceException` if element `xyz` does not exist. If you’d prefer a `null` rather than an exception, cast the `XElement` to a `string` instead of querying its `Value` property. In other words:

```
string xyz = (string) settings.Element ("xyz");
```

This works because `XElement` defines conversion—just for this purpose!

an

an

explicit string

Recursive functions

XContainer also provides Descendants and DescendantNodes methods that return child elements or nodes, *recursively*. Descendants accepts an optional element name. Returning to our earlier example, we can use Descendants to find all the hand tools as follows:

```
Console.WriteLine (bench.Descendants ("handtool").Count());
```

```
// 3
```

Both parent and leaf nodes are included, as the following example demonstrates:

```
foreach (XNode node in bench.DescendantNodes())
  Console.WriteLine (node.ToString (SaveOptions.DisableFormatting));
```

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
```

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

The next query extracts all comments anywhere within the X-DOM that contain the word “careful”:

```
IEnumerable<string> query =
  from c in bench.DescendantNodes().OfType<XComment>()
  where c.Value.Contains ("careful")
  orderby c.Value
  orderby c.Value
  select c.Value;
```

Parent Navigation

All XNodes have a Parent property and Ancestor*XXX* methods for parent navigation. A parent is always an XElement:

Return type	Members	Works on
XElement	Parent { get; }	XNode*
Enumerable<XElement>	Ancestors()	XNode*
	Ancestors (XName)	XNode*
	AncestorsAndSelf()	XElement*
	AncestorsAndSelf (XName)	XElement*

If x is an XElement, the following always prints true:

If x is an XElement, the following always prints true:

```
foreach (XNode child in x.Nodes())
  Console.WriteLine (child.Parent == x);
```

The same is not the case, however, if x is an XDocument. XDocument is peculiar: it can have children, but can never be anyone's parent! To access the XDocument, you instead use the Document property—this works on any object in the X-DOM tree.

Ancestors returns a sequence whose first element is Parent, and whose next element is Parent.Parent, and so on, until the root element.

You can navigate to the root element with the LINQ query AncestorsAndSelf().Last().

Another way to achieve the same thing is to call Document.Root—although this works only if an XDocument is present.

Peer Node Navigation

Peer Node Navigation

Return type	Members	Defined in
bool	IsBefore (XNode node)	XNode

XNode

IEnumerable<XNode>

IsAfter (XNode node)

PreviousNode { get; }

NextNode { get; }

NodesBeforeSelf()

NodesAfterSelf()

NodesBeforeSelf()

NodesAfterSelf()

XNode

XNode

XNode

XNode

XNode

Return type	Members	Defined in
IEnumerable<XElement>	ElementsBeforeSelf()	XNode
	ElementsBeforeSelf (XName name)	XNode
	ElementsAfterSelf()	XNode

	ElementsAfterSelf()	XNode
	ElementsAfterSelf (XName name)	XNode

LINQ to XML

With PreviousNode and NextNode (and FirstNode/LastNode), you can traverse nodes with the feel of a linked list. This is noncoincidental: internally, nodes are stored in a linked list.

XNode internally uses a *singly* linked list, so PreviousNode is not performant.

Attribute Navigation

Return type	Members	Defined in
bool	HasAttributes { get; }	XElement
XAttribute	Attribute (XName name)	XElement
	FirstAttribute { get; }	XElement

	`FirstAttribute { get; }`	`XElement`
	`LastAttribute { get; }`	`XElement`
`IEnumerable<XAttribute>`	`Attributes()`	`XElement`
	`Attributes (XName name)`	`XElement`

In addition, `XAttribute` defines `PreviousAttribute` and `NextAttribute` properties, as well as `Parent`.

The `Attributes` method that accepts a name returns a sequence with either zero or one element; an element cannot have duplicate attribute names in XML.

Updating an X-DOM

You can update elements and attributes in the following ways:

●

●

●

Call SetValue or reassign the Value property.

Call SetElementValue or SetAttributeValue.

Call one of the Remove*XXX* methods.

Call one of the Add*XXX* or Replace*XXX* methods, specifying fresh content.

You can also reassign the Name property on XElement objects.

Simple Value Updates

Members	Works on

Members	Works on
`SetValue (object value)`	`XElement`, `XAttribute`
`Value { get; set }`	`XElement`, `XAttribute`

The SetValue method replaces an element or attribute's content with a simple value. Setting the Value property does the same, but accepts string data only. We describe both of these functions in detail later in this chapter (see the section "Working with Values" on page 428).

An effect of calling SetValue (or reassigning Value) is that it replaces all child nodes:

```
XElement settings = new XElement ("settings",
                      new XElement ("timeout", 30)
                    );
settings.SetValue ("blah");
Console.WriteLine (settings.ToString());  // <settings>blah</settings>
```

Updating Child Nodes and Attributes

Category	Members	Works on

Category	Members	Works on
Add	Add (params object[] content)	XContainer
	AddFirst (params object[] content)	XContainer
Remove	RemoveNodes()	XContainer
	RemoveAttributes()	XElement
	RemoveAll()	XElement
Update	ReplaceNodes (params object[] content)	XContainer
	ReplaceAttributes (params object[] content)	XElement
	ReplaceAll (params object[] content	XElement
	SetElementValue (XName name, object value)	XElement
	SetAttributeValue (XName name, object value)	XElement

The most convenient methods in this group are the last two: SetElementValue and SetAttributeValue. They serve as shortcuts for instantiating an XElement or XAttribute and then Adding it to a parent, replacing any existing element or attribute of that name:

XAttribute and then Adding it to a parent, replacing any existing element or attribute of that name:

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30);     // Adds child node
settings.SetElementValue ("timeout", 60);     // Update it to 60
```

Add appends a child node to an element or document. AddFirst does the same thing, but inserts at the beginning of the collection rather than the end.

You can remove all child nodes or attributes in one hit with RemoveNodes or RemoveAttributes. RemoveAll is equivalent to calling both of these methods.

The Replace*XXX* methods are equivalent to Removing and then Adding. They take a snapshot of the input, so e.ReplaceNodes(e.Nodes()) works as expected.

Updating Through the Parent

Members	Works on
`AddBeforeSelf (params object[] content)`	`XNode`
`AddAfterSelf (params object[] content)`	`XNode`
`Remove()`	`XNode*, XAttribute*`
`ReplaceWith (params object[] content)`	`XNode`

[redacted]

The methods AddBeforeSelf, AddAfterSelf, Remove, and ReplaceWith don't operate on the node's children. Instead, they operate on the collection in which the node itself is in. This requires that the node have a parent element—otherwise, an exception is thrown. AddBeforeSelf and AddAfterSelf are useful for inserting a node into an arbitrary position:

```
XElement items = new XElement ("items",
                   new XElement ("one"),
                   new XElement ("three")
                 );
items.FirstNode.AddAfterSelf (new XElement ("two"));
```

Here's the result:

```
<items><one /><two /><three /></items>
```

Inserting into an arbitrary position within a long sequence of elements is actually quite efficient, because nodes are stored internally in a linked list.

quite efficient, because nodes are stored internally in a linked list.

The Remove method removes the current node from its parent. ReplaceWith does the same—and then inserts some other content at the same position. For instance:

```
XElement items = XElement.Parse ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith (new XComment ("One was here"));
```

Here's the result:

```
<items><!--one was here--><two /><three /></items>
```

Removing a sequence of nodes or attributes

Thanks to extension methods in System.Xml.Linq, you can also call Remove on a *sequence* of nodes or attributes. Consider this X-DOM:

```
XElement contacts = XElement.Parse (
@"<contacts>
    <customer name='Mary'/>
    <customer name='Chris' archived='true'/>
    <supplier name='Susan'>
      <phone archived='true'>012345678<!--confidential--></phone>
    </supplier>
    <supplier name="Susan">
      <phone archived='true'>012345678<!--confidential--></phone>
    </supplier>
  </contacts>");
```

The following removes all customers:

```
contacts.Elements ("customer").Remove();
```

The next statement removes all archived contacts (so *Chris* disappears):

```
contacts.Elements().Where (e => (bool?) e.Attribute ("archived") == true)
                   .Remove();
```

If we replaced Elements() with Descendants(), all archived elements throughout the DOM would disappear, with this result:

```
<contacts>
  <customer name="Mary" />
```