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

The next example removes all contacts that feature the comment “confidential” anywhere in their tree:

```
contacts.Elements().Where (e => e.DescendantNodes()
                                 .OfType<XComment>()
                                 .Any (c => c.Value == "confidential")
                          ).Remove();
```

This is the result:

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

```
</contacts>
```

Contrast this with the following simpler query, which strips all comment nodes from the tree:

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```

Internally, the `Remove` methods first read all matching elements into a temporary list, and then enumerate over the temporary list to perform the deletions. This avoids errors that could otherwise result from deleting and querying at the same time.

# Working with Values

`XElement` and `XAttribute` both have a `Value` property of type `string`. If an element has a single `XText` child node, `XElement`'s `Value` property acts as a convenient shortcut to the content of that node. With `XAttribute`, the `Value` property is simply the attribute's value.

tribute's value.

Despite the storage differences, the X-DOM provides a consistent set of operations for working with element and attribute values.



# Setting Values

There are two ways to assign a value: call `SetValue` or assign the `Value` property. `SetValue` is more flexible because it accepts not just strings, but other simple data types too:

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value);              // 2007-03-02T16:39:10.734375+09:00
```

We could have instead just set the element's `Value` property, but this would mean manually converting the `DateTime` to a string. This is more complicated than calling `ToString`—it requires the use of `XmlConvert` for an XML-compliant result.

When you pass a *value* into `XElement` or `XAttribute`'s constructor, the same automatic conversion takes place for nonstring types. This ensures that `DateTime`s are correctly formatted; `true` is written in lowercase, and `double.NegativeInfinity` is written as "-INF".

# Getting Values

To go the other way around and parse a `Value` back to a base type, you simply cast the `XElement` or `XAttribute` to the desired type. It sounds like it shouldn't work—but it does! For instance:

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;

XAttribute a = new XAttribute ("resolution", 1.234);
double res = (double) a;
```

An element or attribute doesn't store `DateTime`s or numbers natively—they're always stored as text, and then parsed as needed. It also doesn't "remember" the original type, so you must cast it correctly to avoid a runtime error. To make your code robust, you can put the cast in a `try/catch` block, catching a `FormatException`.

Explicit casts on `XElement` and `XAttribute` can parse to the following types:

- 
- 
-

- All standard numeric types
- `string`, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan`, and `Guid`
- `Nullable<>` versions of the aforementioned value types

Casting to a nullable type is useful in conjunction with the `Element` and `Attribute` methods, because if the requested name doesn't exist, the cast still works. For instance, if `x` has no `timeout` element, the first line generates a runtime error and the second line does not:

```
int timeout = (int) x.Element ("timeout");
int? timeout = (int?) x.Element ("timeout");
```

```
// Error
// OK; timeout is null.
```

You can factor away the nullable type in the final result with the ?? operator. The following evaluates to 1.0 if the resolution attribute doesn't exist:

```
double resolution = (double?) x.Attribute ("resolution") ?? 1.0;
```



Casting to a nullable type won't get you out of trouble, though, if the element or attribute *exists* and has an empty (or improperly formatted) value. For this, you must catch a FormatException.

You can also use casts in LINQ queries. The following returns "John":

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100' />
      <customer id='2' name='John' credit='150' />
      <customer id='3' name='Anne' />
    </data>");

IEnumerable<string> query = from cust in data.Elements()
                            where (int?) cust.Attribute ("credit") > 100
                            select cust.Attribute ("name").Value;
```

```
                                select cust.Attribute ("name").Value;
```

Casting to a nullable `int` avoids a `NullReferenceException` in the case of Anne, who has no `credit` attribute. Another solution would be to add a predicate to the `where` clause:

```
where cust.Attributes ("credit").Any() && (int) cust.Attribute...
```

The same principles apply in querying element values.

# Values and Mixed Content Nodes

Giventhe value of `Value`, you might wonder when you'd ever need to deal directly with `XText` nodes. The answer is when you have mixed content. For example:

```
<summary>An XAttribute is <bold>not</bold> an XNode</summary>
```

A simple `Value` property is not enough to capture `summary`'s content. The `summary` element contains three children: an `XText` node followed by an `XElement`, followed by another `XText` node. Here's how to construct it:

```
XElement summary = new XElement ("summary",
                        new XText ("An XAttribute is "),
                        new XElement ("bold", "not"),
```

```
                    new XText ("An XAttribute is "),
                    new XElement ("bold", "not"),
                    new XText (" an XNode")
                  );
```

Interestingly, we can still query `summary`'s `Value`—without getting an exception. Instead, we get a concatenation of each child's value:

```
An XAttribute is not an XNode
```

It's also legal to reassign `summary`'s `Value`, at the cost of replacing all previous children with a single new `XText` node.

# Automatic XText Concatenation

When you add simple content to an `XElement`, the X-DOM appends to the existing `XText` child rather than creating a new one. In the following examples, `e1` and `e2` end up with just one child `XText` element whose value is `HelloWorld`:

```
var e1 = new XElement ("test", "Hello"); e1.Add ("World");
var e2 = new XElement ("test", "Hello", "World");
```

If you specifically create XText nodes, however, you end up with multiple children:

```
var e = new XElement ("test", new XText ("Hello"), new XText ("World"));
Console.WriteLine (e.Value);              // HelloWorld
Console.WriteLine (e.Nodes().Count());    // 2
```

`XElement` doesn't concatenate the two `XText` nodes, so the nodes' object identities are preserved.

# Documents and Declarations

## XDocument

As we said previously, an `XDocument` wraps a root `XElement` and allows you to add an `XDeclaration`, processing instructions, a document type, and root-level comments. An `XDocument` is optional and can be ignored or omitted: unlike with the W3C DOM, it does not serve as glue to keep everything together.

An `XDocument` provides the same functional constructors as `XElement`. And because it's based on `XContainer`, it also supports the `Add`*XXX*, `Remove`*XXX*, and `Replace`*XXX* methods. Unlike `XElement`, however, an `XDocument` can accept only limited content:

- :

- A single `XElement` object (the "root")
- A single `XDeclaration` object
- A single `XDocumentType` object (to reference a DTD)
- Any number of `XProcessingInstruction` objects
- Any number of `XComment` objects

Of these, only the root `XElement` is mandatory in order to have a valid `XDocument`. The `XDeclaration` is optional—if omitted, default settings are applied during serialization.

The simplest valid XDocument has just a root element:

```
var doc = new XDocument (
            new XElement ("test", "data")
          );
```

Notice that we didn't include an XDeclaration object. The file generated by calling doc.Save would still contain an XML declaration, however, because one is generated by default.



The next example produces a simple but correct XHTML file, illustrating all the constructs that an XDocument can accept:

```
var styleInstruction = new XProcessingInstruction (
  "xml-stylesheet", "href='styles.css' type='text/css'");

var docType = new XDocumentType ("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
```

```
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);
```

```
XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement (ns + "html",
    new XElement (ns + "head",
      new XElement (ns + "title", "An XHTML page")),
    new XElement (ns + "body",
      new XElement (ns + "p", "This is the content"))
  );
var doc =
  new XDocument (
    new XDeclaration ("1.0", "utf-8", "no"),
    new XComment ("Reference a stylesheet"),
    styleInstruction,
    docType,
    root);
```

```
    doctype,
    root);
doc.Save ("test.html");
```

The resultant *test.html* reads as follows:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
  <body>
    <p>This is the content</p>
  </body>
</html>
```

`XDocument` has a `Root` property that serves as a shortcut for accessing a document's single `XElement`. The reverse link is provided by `XObject`'s `Document` property, which

XDocument has a Root property that serves as a shortcut for accessing a document's single XElement. The reverse link is provided by XObject's Document property, which works for all objects in the tree:

```
Console.WriteLine (doc.Root.Name.LocalName);          // html
XElement bodyNode = doc.Root.Element (ns + "body");
Console.WriteLine (bodyNode.Document == doc);         // True
```

Recall that a document's children have no Parent:



```
Console.WriteLine (doc.Root.Parent == null);
foreach (XNode node in doc.Nodes())
  Console.Write (node.Parent == null);

// True
// TrueTrueTrueTrue
```

An `XDeclaration` is not an `XNode` and does not appear in the document's `Nodes` collection—unlike comments, processing instructions, and the root element. Instead, it gets assigned to a dedicated property called `Declaration`. This is why "True" is repeated four and not five times in the last example.



# XML Declaration

# XML Declarations

A standard XML file starts with a declaration such as the following:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

An XML declaration ensures that the file will be correctly parsed and understood by a reader. `XElement` and `XDocument` follow these rules in emitting XML declarations:

- 
- 
- 

Calling `Save` with a filename always writes a declaration.

Calling `Save` with an `XmlWriter` writes a declaration unless the `XmlWriter` is instructed otherwise.

The `ToString` method never emits an XML declaration.

You can instruct an `XmlWriter` not to produce a declaration by setting the `OmitXmlDeclaration` and `ConformanceLevel` properties of an `XmlWriterSettings` object when constructing the `XmlWriter`. We describe this in Chapter 11.

The presence or absence of an `XDeclaration` object has no effect on whether an XML declaration gets written. The purpose of an `XDeclaration` is instead to *hint the XML serialization*—in two ways:

- What text encoding to use
- What to put in the XML declaration's `encoding` and `standalone` attributes (should a declaration be written)

`XDeclaration`'s constructor accepts three arguments, which correspond to the attributes `version`, `encoding`, and `standalone`. In the following example, *test.xml* is encoded in UTF-16:

encoded in UTF-16:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-16", "yes"),
            new XElement ("test", "data")
          );
doc.Save ("test.xml");
```



Whatever you specify for the XML version is ignored by the XML writer: it always writes `"1.0"`.

The encoding must use an IETF code such as `"utf-16"`—just as it would appear in the XML declaration.

# Writing a declaration to a string

# Writing a declaration to a string

Suppose we want to serialize an `XDocument` to a `string`—including the XML declaration. Because `ToString` doesn't write a declaration, we'd have to use an `XmlWriter` instead:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-8", "yes"),
            new XElement ("test", "data")
          );
var output = new StringBuilder();
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create (output, settings))
  doc.Save (xw);
Console.WriteLine (output.ToString());
```

This is the result:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

Notice that we got UTF-16 in the output—even though we explicitly requested UTF-8 in an `XDeclaration`! This might look like a bug, but in fact, `XmlWriter` is being remarkably smart. Because we're writing to a `string` and not a file or stream, it's impossible to apply any encoding other than UTF-16—the format in which strings are internally stored. Hence, `XmlWriter` writes `"utf-16"`—so as not to lie.

This also explains why the `ToString` method doesn't emit an XML declaration. Imagine that instead of calling `Save`, you did the following to write an `XDocument` to a file:

```
File.WriteAllText ("data.xml", doc.ToString());
```

As it stands, *data.xml* would lack an XML declaration, making it incomplete but still parsable (you can infer the text encoding). But if `ToString()` emitted an XML declaration, *data.xml* would actually contain an *incorrect* declaration (`encoding="utf-16"`), which might prevent it from being read at all, because `WriteAllText` encodes using UTF-8.

# Names and Namespaces

# Names and Namespaces

Just as .NET types can have namespaces, so too can XML elements and attributes.

XML namespaces achieve two things. First, rather like namespaces in C#, they help avoid naming collisions. This can become an issue when you merge data from one



XML file into another. Second, namespaces assign *absolute* meaning to a name. The name "nil," for instance, could mean anything. Within the *http://www.w3.org/2001/xmlschema-instance* namespace, however, "nil" means something equivalent to `null` in C# and comes with specific rules on how it can be applied.

Because XML namespaces are a significant source of confusion, we'll cover the topic first in general, and then move on to how they're used in LINQ to XML.

# Namespaces in XML

Suppose we want to define a `customer` element in the namespace `OReilly.Nutshell.CSharp`. There are two ways to proceed. The first is to use the `xmlns` attribute as follows:

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

`xmlns` is a special reserved attribute. When used in this manner, it performs two functions:

- It specifies a namespace for the element in question.
- It specifies a default namespace for all descendant elements.

This means that in the following example, `address` and `postcode` implicitly live in the `OReilly.Nutshell.CSharp` namespace:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

```
</customer>
```

If we want address and postcode to have *no* namespace, we'd have to do this:

```
<customer xmlns="OReilly.Nutshell.CSharp"s>
  <address xmlns="">
    <postcode>02138</postcode>     <!-- postcode now inherits empty ns -->
  </address>
</customer>
```

# Prefixes

The other way to specify a namespace is with a *prefix*. A prefix is an alias that you assign to a namespace to save typing. There are two steps in using a prefix—*defining* the prefix and *using* it. You can do both together as follows:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp"/>
```

Two distinct things are happening here. On the right, xmlns:nut="..." defines a prefix called nut and makes it available to this element and all its descendants. On the left, nut:customer assigns the newly allocated prefix to the customer element.

the left, `nut:customer` assigns the newly allocated prefix to the `customer` element.

A prefixed element *does not* define a default namespace for descendants. In the following XML, `firstname` has an empty namespace:



```
<nut:customer nut:xmlns="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

To give `firstname` the `OReilly.Nutshell.CSharp` prefix, we must do this:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

You can also define a prefix—or prefixes—for the convenience of your descendants, without assigning any of them to the parent element itself. The following defines two prefixes, `i` and `z`, while leaving the `customer` element itself with an empty namespace:

two prefixes, i and z, while leaving the customer element itself with an empty namespace:

```
<customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  ...
</customer>
```

If this was the root node, the whole document would have i and z at its fingertips. Prefixes are convenient when elements need to draw from a number of namespaces.

Notice that both namespaces in this example are URIs. Using URIs (that you own) is standard practice: it ensures namespace uniqueness. So, in real life, our customer element would more likely be:

```
<customer xmlns="http://oreilly.com/schemas/nutshell/csharp"/>
```

or:

```
<nut:customer xmlns:nut="http://oreilly.com/schemas/nutshell/csharp"/>
```

# Attributes

You can assign namespaces to attributes too. The main difference is that it always

You can assign namespaces to attributes too. The main difference is that it always requires a prefix. For instance:

```
<customer xmlns:nut="OReilly.Nutshell.CSharp" nut:id="123" />
```

Another difference is that an unqualified attribute always has an empty namespace: it never inherits a default namespace from a parent element.

Attributes tend not to need namespaces because their meaning is usually local to the element. An exception is with general-purpose or metadata attributes, such as the `nil` attribute defined by W3C:

```
<customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstname>Joe</firstname>
  <lastname xsi:nil="true"/>
</customer>
```

This indicates unambiguously that `lastname` is `nil` (`null` in C#) and not an empty string. Because we've used the standard namespace, a general-purpose parsing utility could know with certainty our intention.

# Specifying Namespaces in the X-DOM

So far in this chapter, we've used just simple strings for `XElement` and `XAttribute` names. A simple string corresponds to an XML name with an empty namespace—rather like a .NET type defined in the global namespace.

There are a couple of ways to specify an XML namespace. The first is to enclose it in braces, before the local name. For example:

```
var e = new XElement ("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine (e.ToString());
```

Here's the resulting XML:

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

The second (and more performant) approach is to use the `XNamespace` and `XName` types. Here are their definitions:

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}
```

```
public sealed class XName     // A local name with optional namespace
```

```
public sealed class XName     // A local name with optional namespace
{
  public string LocalName { get; }
  public XNamespace Namespace { get; }   // Optional
}
```

Both types define implicit casts from `string`, so the following is legal:

```
XNamespace ns   = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  = "{http://domain.com/xmlspace}customer";
```

`XName` also overloads the `+` operator, allowing you to combine a namespace and name without using braces:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName);     // {http://domain.com/xmlspace}customer
```

All constructors and methods in the X-DOM that accept an element or attribute name actually accept an `XName` object rather than a `string`. The reason you can substitute a string—as in all our examples to date—is because of the implicit cast.

stitute a string—as in all our examples to date—is because of the implicit cast.

Specifying a namespace is the same whether for an element or an attribute:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
                 new XAttribute (ns + "id", 123)
             );
```



# The X-DOM and Default Namespaces

The X-DOM ignores the concept of default namespaces until it comes time to actually output XML. This means that when you construct a child `XElement`, you must give it a namespace explicitly if needed: it *will not* inherit from the parent:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
               new XElement (ns + "customer", "Bloggs"),
               new XElement (ns + "purchase", "Bicycle")
            );
```

The X-DOM does, however, apply default namespaces when reading and outputting

The X-DOM does, however, apply default namespaces when reading and outputting XML:

```
Console.WriteLine (data.ToString());
OUTPUT:
  <data xmlns="http://domain.com/xmlspace">
    <customer>Bloggs</customer>
    <purchase>Bicycle</purchase>
  </data>
```

```
Console.WriteLine (data.Element (ns + "customer").ToString());
```

```
OUTPUT:
  <customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

If you construct `XElement` children without specifying namespaces—in other words:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data"
```

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement ("customer", "Bloggs"),
             new XElement ("purchase", "Bicycle")
           );
Console.WriteLine (data.ToString());
```

you get this result instead:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

Another trap is failing to include a namespace when navigating an X-DOM:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer", "Bloggs"),
             new XElement (ns + "purchase", "Bicycle")
           );
XElement x = data.Element (ns + "customer");    // ok
```

```
            );
XElement x = data.Element (ns + "customer");    // ok
XElement y = data.Element ("customer");         // null
```

If you build an X-DOM tree without specifying namespaces, you can subsequently assign every element to a single namespace as follows:



```
foreach (XElement e in data.DescendantsAndSelf())
  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

# Prefixes

The X-DOM treats prefixes just as it treats namespaces: purely as a serialization function. This means you can choose to completely ignore the issue of prefixes—and get by! The only reason you might want to do otherwise is for efficiency when outputting to an XML file. For example, consider this:

```
XNamespace ns1 = "http://domain.com/space1";
```

```
XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";
```

# LINQ to XML

```
var mix = new XElement (ns1 + "data",
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value"),
            new XElement (ns2 + "element", "value")
        );
```

```
            new XElement (ns2 + "element", "value")
        );
```

By default, `XElement` will serialize this as follows:

```
<data xmlns="http://domain.com/space1">
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
</data>
```

As you can see, there's a bit of unnecessary duplication. The solution is *not* to change the way you construct the X-DOM, but instead to give the serializer a hint prior to writing the XML. Do this by adding attributes defining prefixes that you want to see applied. This is typically done on the root element:

```
mix.SetAttributeValue (XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2", ns2);
```

This assigns the prefix "ns1" to our `XNamespace` variable `ns1`, and "ns2" to `ns2`. The X-DOM automatically picks up these attributes when serializing and uses them to condense the resulting XML. Here's the result now of calling `ToString` on `mix`:

condense the resulting XML. Here's the result now of calling `ToString` on `mix`:

```
<ns1:data xmlns:ns1="http://domain.com/space1"
          xmlns:ns2="http://domain.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Prefixes don't change the way you construct, query, or update the X-DOM—for these activities, you ignore the presence of prefixes and continue to use full names. Prefixes come into play only when converting to and from XML files or streams.

Prefixes are also honored in serializing attributes. In the following example, we record a customer's date of birth and credit as `"nil"`, using the W3C-standard attribute. The highlighted line ensures that the prefix is serialized without unnecessary namespace repetition:



```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);
```

```
var cust = new XElement ("customers",
             new XAttribute (XNamespace.Xmlns + "xsi", xsi),
             new XElement ("customer",
               new XElement ("lastname", "Bloggs"),
               new XElement ("dob", nil),
               new XElement ("credit", nil)
             )
           );
```

# This is its XML:

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

```
</customers>
```

For brevity, we predeclared the nil `XAttribute` so that we could use it twice in building the DOM. You're allowed to reference the same attribute twice because it's automatically duplicated as required.

# Annotations

You can attach custom data to any `XObject` with an annotation. Annotations are intended for your own private use and are treated as black boxes by X-DOM. If you've ever used the `Tag` property on a Windows Forms or WPF control, you'll be familiar with the concept—the difference is that you have multiple annotations, and your annotations can be *privately scoped*. You can create an annotation that other types cannot even see—let alone overwrite.

The following methods on `XObject` add and remove annotations:

```
public void AddAnnotation (object annotation)
public void RemoveAnnotations<T>()     where T : class
```

The following methods retrieve annotations:

The following methods retrieve annotations:

```
public T Annotation<T>()                    where T : class
public IEnumerable<T> Annotations<T>() where T : class
```

Each annotation is keyed by its *type*, which must be a reference type. The following adds and then retrieves a `string` annotation:

```
XElement e = new XElement ("test");
e.AddAnnotation ("Hello");
Console.WriteLine (e.Annotation<string>());   // Hello
```

You can add multiple annotations of the same type, and then use the `Annotations` method to retrieve a *sequence* of matches.



A public type such as `string` doesn't make a great key, however, because code in other types can interfere with your annotations. A better approach is to use an internal or (nested) private class:

```
class X
```

```
class X
{
  class CustomData { internal string Message; }

// Private nested type

LINQ to XML

static void Test()
```

```
  static void Test()
  {
    XElement e = new XElement ("test");
    e.AddAnnotation (new CustomData { Message = "Hello" } );
    Console.Write (e.Annotations<CustomData>().First().Message);
  }
}
```

```
// Hello
```

To remove annotations, you must also have access to the key's type:

```
e.RemoveAnnotations<CustomData>();
```

# Projecting into an X-DOM

So far, we've shown how to use LINQ to get data *out* of an X-DOM. You can also use LINQ queries to project *into* an X-DOM. The source can be anything over which

use LINQ queries to project *into* an X-DOM. The source can be anything over which LINQ can query, such as:

- LINQ to SQL or Entity Framework queries
- A local collection
- Another X-DOM

Regardless of the source, the strategy is the same in using LINQ to emit an X-DOM: first write a *functional construction* expression that produces the desired X-DOM shape, and then build a LINQ query around the expression.

For instance, suppose we want to retrieve customers from a database into the following XML:

```
<customers>
```

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

We start by writing a functional construction expression for the X-DOM using simple literals:

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute ("id", 1),
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
```

```
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
    )
  );
```



We then turn this into a projection and build a LINQ query around it:

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
```

In Entity Framework, you must call `.ToList()` after retrieving customers, so that the third line reads:

```
from c in objectContext.Customers.ToList()
```

Here's the result:

```
<customers>
  <customer id="1">
    <name>Tom</name>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</name>
```

```
    <name>Harry</name>
    <buys>2</buys>
  </customer>
    ...
</customers>
```

We can see how this works more clearly by constructing the same query in two steps. First:

```
IEnumerable<XElement> sqlQuery =
  from c in dataContext.Customers
  select
    new XElement ("customer", new XAttribute ("id", c.ID),
      new XElement ("name", c.Name),
      new XElement ("buys", c.Purchases.Count)
    );
```

This inner portion is a normal LINQ to SQL query that projects into custom types

This inner portion is a normal LINQ to SQL query that projects into custom types (from LINQ to SQL's perspective). Here's the second step:

```
var customers = new XElement ("customers", sqlQuery);
```

This constructs the root `XElement`. The only thing unusual is that the content, `sqlQuery`, is not a single `XElement` but an `IQueryable<XElement>`—which implements `IEnumerable<XElement>`. Remember that in the processing of XML content, collections are automatically enumerated. So, each `XElement` gets added as a child node.

This outer query also defines the line at which the query transitions from being a database query to a local LINQ to enumerable query. `XElement`'s constructor doesn't



know about `IQueryable<>`, so it forces enumeration of the database query—and execution of the SQL statement.

# Eliminating Empty Elements

Suppose in the preceding example that we also wanted to include details of the

Suppose in the preceding example that we also wanted to include details of the customer's most recent high-value purchase. We could do this as follows:



```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
```

```
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count),
        new XElement ("lastBigBuy",
          new XElement ("description",
            lastBigBuy == null ? null : lastBigBuy.Description),
          new XElement ("price",
            lastBigBuy == null ? 0m : lastBigBuy.Price)
        )
      )
  );
```

This emits empty elements, though, for customers with no high-value purchases. (If it was a local query rather than a database query, it would throw a `NullReferenceException`.) In such cases, it would be better to omit the `lastBigBuy` node entirely. We can achieve this by wrapping the constructor for the `lastBigBuy` element in a conditional operator:

can achieve this by wrapping the constructor for the lastBigBuy element in a conditional operator:

```
select
  new XElement ("customer", new XAttribute ("id", c.ID),
    new XElement ("name", c.Name),
    new XElement ("buys", c.Purchases.Count),
    lastBigBuy == null ? null :
      new XElement ("lastBigBuy",
        new XElement ("description", lastBigBuy.Description),
        new XElement ("price", lastBigBuy.Price)
```

For customers with no lastBigBuy, a null is emitted instead of an empty XElement. This is what we want, because null content is simply ignored.

# Streaming a Projection

If you're projecting into an X-DOM only to Save it (or call ToString on it), you can improve memory efficiency through an XStreamingElement. An XStreamingElement is a cut-down version of XElement that applies *deferred loading* semantics to its child content. To use it, you simply replace the outer XElements with XStreamingElements:

```
var customers =
  new XStreamingElement ("customers",
    from c in dataContext.Customers
    select
      new XStreamingElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
customers.Save ("data.xml");
```

The queries passed into an `XStreamingElement`'s constructor are not enumerated until you call `Save`, `ToString`, or `WriteTo` on the element; this avoids loading the whole X-DOM into memory at once. The flipside is that the queries are reevaluated, should you re-`Save`. Also, you cannot traverse an `XStreamingElement`'s child content—it does not expose methods such as `Elements` or `Attributes`.

`XStreamingElement` is not based on `XObject`—or any other class—because it has such a limited set of members. The only members it has, besides `Save`, `ToString`, and

XStreamingElement is not based on XObject—or any other class—because it has such a limited set of members. The only members it has, besides Save, ToString, and WriteTo, are an Add method, which accepts content like the constructor and a Name property.

XStreamingElement does not allow you to *read* content in a streamed fashion—for this, you must use an XmlReader in conjunction with the X-DOM. We describe how to do this in the section "Patterns for Using XmlReader/XmlWriter" on page 459 in Chapter 11.

# Transforming an X-DOM

You can transform an X-DOM by reprojecting it. For instance, suppose we want to transform an *msbuild* XML file, used by the C# compiler and Visual Studio to describe a project, into a simple format suitable for generating a report. An *msbuild* file looks like this:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/dev...>
  <PropertyGroup>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>9.0.11209</ProductVersion>
    ...
```

```
  <ProductVersion>9.0.11209</ProductVersion>
  ...
</PropertyGroup>
<ItemGroup>
  <Compile Include="ObjectGraph.cs" />
  <Compile Include="Program.cs" />
  <Compile Include="Properties\AssemblyInfo.cs" />
  <Compile Include="Tests\Aggregation.cs" />
  <Compile Include="Tests\Advanced\RecursiveXml.cs" />
</ItemGroup>
<ItemGroup>
```

```
    ...
  </ItemGroup>
  ...
</Project>
```

Let's say we want to include only files, as follows:

```
<ProjectReport>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <File>Properties\AssemblyInfo.cs</File>
  <File>Tests\Aggregation.cs</File>
  <File>Tests\Advanced\RecursiveXml.cs</File>
</ProjectReport>
```

The following query performs this transformation:

```
XElement project = XElement.Load ("myProjectFile.csproj");
XNamespace ns = project.Name.Namespace;
var query =
  new XElement ("ProjectReport",
    from compileItem in
      project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
    let include = compileItem.Attribute ("Include")
    where include != null
    select new XElement ("File", include.Value)
  );
```

The query first extracts all `ItemGroup` elements, and then uses the `Elements` extension method to obtain a flat sequence of all their `Compile` subelements. Notice that we had to specify an XML namespace—everything in the original file inherits the namespace defined by the `Project` element—so a local element name such as `ItemGroup`

had to specify an XML namespace—everything in the original file inherits the namespace defined by the `Project` element—so a local element name such as `ItemGroup` won't work on its own. Then, we extracted the `Include` attribute value and projected its value as an element.

# Advanced transformations

When querying a local collection such as an X-DOM, you're free to write custom query operators to assist with more complex queries.

Suppose in the preceding example that we instead wanted a hierarchical output, based on folders:

```
<Project>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <Folder name="Properties">
    <File>AssemblyInfo.cs</File>
```

```
    <File>AssemblyInfo.cs</File>
  </Folder>
  <Folder name="Tests">
    <File>Aggregation.cs</File>
    <Folder name="Advanced">
      <File>RecursiveXml.cs</File>
    </Folder>
  </Folder>
</Project>
```

To produce this, we need to process path strings such as *Tests\Advanced\RecursiveXml.cs* recursively. The following method does just this: it accepts a sequence of path strings and emits an X-DOM hierarchy consistent with our desired output:

path strings and emits an X-DOM hierarchy consistent with our desired output.



```
static IEnumerable<XElement> ExpandPaths (IEnumerable<string> paths)
{
  var brokenUp = from path in paths
                 let split = path.Split (new char[] { '\\' }, 2)
                 orderby split[0]
                 select new
                 {
                   name = split[0],
                   remainder = split.ElementAtOrDefault (1)
                 };

IEnumerable<XElement> files = from b in brokenUp
                              where b.remainder == null
                              select new XElement ("file", b.name);

  IEnumerable<XElement> folders = from b in brokenUp
                                  where b.remainder != null
                                  group b.remainder by b.name into grp
                                  select new XElement ("folder",
```

```
                                        group b.remainder by b.name into grp
                                        select new XElement ("folder",
                                          new XAttribute ("name", grp.Key),
                                          ExpandPaths (grp)
                                        );
  return files.Concat (folders);
}
```

The first query splits each path string at the first backslash, into a `name` + `remainder`:

```
Tests\Advanced\RecursiveXml.cs -> Tests + Advanced\RecursiveXml.cs
```

If `remainder` is `null`, we're dealing with a straight filename. The `files` query extracts these cases.

If `remainder` is not `null`, we've got a folder. The `folders` query handles these cases. Because other files can be in the same folder, it must `group` by folder name to bring them all together. For each group, it then executes the same function for the sub-elements.

The final result is a concatenation of `files` and `folders`. The `Concat` operator preserves order, so all the files come first, alphabetically, then all the folders, alphabetically.

alphabetically.

With this method in place, we can complete the query in two steps. First, we extract a simple sequence of path strings:

```
IEnumerable<string> paths =
  from compileItem in
    project.Elements (ns + "ItemGroup").Elements (ns + "Compile")
  let include = compileItem.Attribute ("Include")
  where include != null
  select include.Value;
```

Then, we feed this into our `ExpandPaths` method for the final result:

```
var query = new XElement ("Project", ExpandPaths (paths));
```

# 11

# Other XML Technologies

The `System.Xml` namespace comprises the following namespaces and core classes:

`System.Xml.*`

`XmlReader` *and* `XmlWriter`
: High-performance, forward-only cursors for reading or writing an XML stream

`XmlDocument`
: Represents an XML document in a W3C-style DOM

`System.Xml.XPath`
: Infrastructure and API (`XPathNavigator`) for XPath, a string-based language for querying XML

`System.Xml.XmlSchema`
: Infrastructure and API for (W3C) XSD schemas

`System.Xml.Xsl`
: Infrastructure and API (`XslCompiledTransform`) for performing (W3C) XSLT transformations of XML

`System.Xml.Serialization`

`System.Xml.Serialization`
: Supports the serialization of classes to and from XML (see Chapter 16)

`System.Xml.XLinq`
: Modern, simplified, LINQ-centric version of `XmlDocument` (see Chapter 10)

W3C is an abbreviation for World Wide Web Consortium, where the XML standards are defined.

`XmlConvert`, the static class for parsing and formatting XML strings, is covered in Chapter 6.

# 447

# XmlReader

`XmlReader` is a high-performance class for reading an XML stream in a low-level, forward-only manner.

Consider the following XML file:

Consider the following XML file:

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

To instantiate an `XmlReader`, you call the static `XmlReader.Create` method, passing in a `Stream`, a `TextReader`, or a URI string. For example:

```
using (XmlReader reader = XmlReader.Create ("customer.xml"))
  ...
```

To construct an `XmlReader` that reads from a string:

```
XmlReader reader = XmlReader.Create (
  new System.IO.StringReader (myString));
```

You can also pass in an `XmlReaderSettings` object to control parsing and validation options. The following three properties on `XmlReaderSettings` are particularly useful for skipping over superfluous content:

options. The following three properties on XmlReaderSettings are particularly useful for skipping over superfluous content:

```
bool IgnoreComments                    // Skip over comment nodes?
bool IgnoreProcessingInstructions      // Skip over processing instructions?
bool IgnoreWhitespace                  // Skip over whitespace?
```

In the following example, we instruct the reader not to emit whitespace nodes, which are a distraction in typical scenarios:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
```

```
using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
  ...
```

Another useful property on XmlReaderSettings is ConformanceLevel. Its default value of Document instructs the reader to assume a valid XML document with a single root node. This is a problem if you want to read just an inner portion of XML, containing multiple nodes:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

To read this without throwing an exception, you must set ConformanceLevel to

To read this without throwing an exception, you must set `ConformanceLevel` to `Fragment`.

`XmlReaderSettings` also has a property called `CloseInput`, which indicates whether to close the underlying stream when the reader is closed (there's an analogous property on `XmlWriterSettings` called `CloseOutput`). The default value for `CloseInput` and `CloseOutput` is `true`.



# Reading Nodes

The units of an XML stream are *XML nodes*. The reader traverses the stream in textual (depth-first) order. The `Depth` property of the reader returns the current depth of the cursor.

The most primitive way to read from an `XmlReader` is to call `Read`. It advances to the next node in the XML stream, rather like `MoveNext` in `IEnumerator`. The first call to `Read` positions the cursor at the first node. When `Read` returns `false`, it means the cursor has advanced *past* the last node, at which point the `XmlReader` should be closed and abandoned.

cursor has advanced *past* the last node, at which point the `XmlReader` should be closed and abandoned.

In this example, we read every node in the XML stream, outputting each node type as we go:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
```



```
using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
```

```
using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
  while (reader.Read())
  {
    Console.Write (new string (' ',reader.Depth*2));  // Write indentation
    Console.WriteLine (reader.NodeType);
  }
```

The output is as follows:

```
XmlDeclaration
Element
  Element
    Text
  EndElement
  Element
    Text
```

```
Text
EndElement
EndElement
```

Attributes are not included in `Read`-based traversal (see the section "Reading Attributes" on page 454, later in this chapter).

`NodeType` is of type `XmlNodeType`, which is an enum with these members:

```
None
XmlDeclaration
Element
EndElement
Text
```

Text
Attribute

Comment
Entity
EndEntity
EntityReference
ProcessingInstruction
CDATA

Document
DocumentType
DocumentFragment

DocumentFragment

Notation

Whitespace

SignificantWhitespace



Two string properties on XmlReader provide access to a node's content: Name and Value. Depending on the node type, either Name or Value (or both) is populated:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.ProhibitDtd = false;      // Must set this to read DTDs

using (XmlReader r = XmlReader.Create ("customer.xml", settings))
  while (r.Read())
  {
    Console.Write (r.NodeType.ToString().PadRight (17, '-'));
    Console.Write ("> ".PadRight (r.Depth * 3));
```

```
Console.Write ("> ".PadRight (r.Depth * 3));
```

```
switch (r.NodeType)
{
  case XmlNodeType.Element:
  case XmlNodeType.EndElement:
    Console.WriteLine (r.Name); break;
  case XmlNodeType.Text:
  case XmlNodeType.CDATA:
  case XmlNodeType.Comment:
  case XmlNodeType.XmlDeclaration:
    Console.WriteLine (r.Value); break;
```

```
case XmlNodeType.DocumentType:
```

```
case XmlNodeType.DocumentType:
  Console.WriteLine (r.Name + " - " + r.Value); break;

    default: break;
  }
}
```

To demonstrate this, we'll expand our XML file to include a document type, entity, CDATA, and comment:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE customer [ <!ENTITY tc "Top Customer"> ]>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <quote><![CDATA[C#'s operators include: < > &]]></quote>
  <notes>Jim Bo is a &tc;</notes>
  <!--  That wasn't so bad! -->
```

```
  <!--  That wasn't so bad! -->
</customer>
```

An entity is like a macro; a CDATA is like a verbatim string (`@"..."`) in C#. Here's the result:

```
XmlDeclaration--> version="1.0" encoding="utf-8"
DocumentType-----> customer -  <!ENTITY tc "Top Customer">
Element----------> customer
Element---------->  firstname
Text-------------->    Jim
EndElement------->  firstname
Element---------->  lastname
Text-------------->    Bo
EndElement------->  lastname
```



```
Element---------->
```

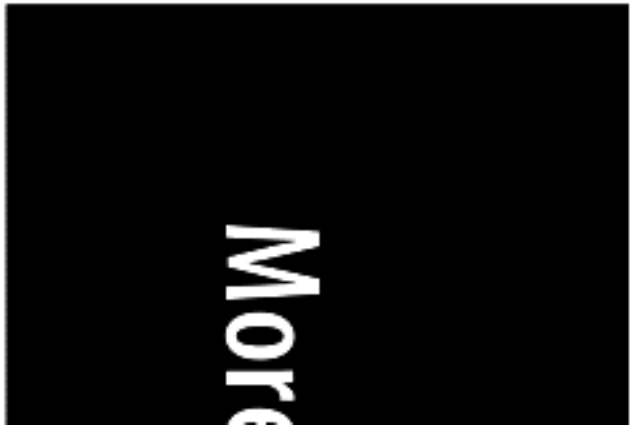
```
Element----------->
CDATA------------->
EndElement-------->
Element----------->
Text-------------->
EndElement-------->
Comment----------->
EndElement-------->
```

```
quote
   C#'s operators include: < > &
quote
notes
```

```
notes
   Jim Bo is a Top Customer
notes
  That wasn't so bad!
customer
```

`XmlReader` automatically resolves entities, so in our example, the entity reference `&tc;` expands into `Top Customer`.

# Reading Elements



Often, you already know the structure of the XML document that you're reading. To help with this, `XmlReader` provides a range of methods that read while *presuming* a particular structure. This simplifies your code, as well as performing some validation at the same time.

`XmlReader` throws an `XmlException` if any validation fails. `XmlException` has `LineNumber` and `LinePosition` properties indicating where the error occurred—logging this information is essential if the XML file is large!

`ReadStartElement` verifies that the current `NodeType` is `StartElement`, and then calls `Read`. If you specify a name, it verifies that it matches that of the current element.

ReadStartElement verifies that the current NodeType is StartElement, and then calls Read. If you specify a name, it verifies that it matches that of the current element.

ReadEndElement verifies that the current NodeType is EndElement, and then calls Read.

For instance, we could read this:

```
<firstname>Jim</firstname>
```

as follows:

```
reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.ReadEndElement();
```

The ReadElementContentAsString method does all of this in one hit. It reads a start element, a text node, and an end element, returning the content as a string:

```
string firstName = reader.ReadElementContentAsString ("firstname", "");
```

```
    string firstName = reader.ReadElementContentAsString ("firstname", "");
```

The second argument refers to the namespace, which is blank in this example. There are also typed versions of this method, such as `ReadElementContentAsInt`, which parse the result. Returning to our original XML document:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit>    <!-- OK, we sneaked this in! -->
</customer>
```



We could read it in as follows:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader r = XmlReader.Create ("customer.xml", settings))
```

```
using (XmlReader r = XmlReader.Create ("customer.xml", settings))
{
  r.MoveToContent();                    // Skip over the XML declaration
  r.ReadStartElement ("customer");
  string firstName    = r.ReadElementContentAsString ("firstname", "");
  string lastName     = r.ReadElementContentAsString ("lastname", "");
  decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

```
    r.MoveToContent();
    r.ReadEndElement();
  }
```

```
  // Skip over that pesky comment
  // Read the closing customer tag
```

The `MoveToContent` method is really useful. It skips over all the

The `MoveToContent` method is really useful. It skips over all the fluff: XML declarations, whitespace, comments, and processing instructions. You can also instruct the reader to do most of this automatically through the properties on `XmlReaderSettings`.

### Optional elements

In the previous example, suppose that `<lastname>` was optional. The solution to this is straightforward:

```
r.ReadStartElement ("customer");
string firstName    = r. ReadElementContentAsString ("firstname", "");
string lastName     = r.Name == "lastname"
                      ? r.ReadElementContentAsString() : null;
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

# Random element order

The examples in this section rely on elements appearing in the XML file in a set order. If you need to cope with elements appearing in any order, the easiest solution is to read that section of the XML into an X-DOM. We describe how to do this later in the section "Patterns for Using XmlReader/XmlWriter" on page 459.

in the section "Patterns for Using XmlReader/XmlWriter" on page 459.

### Empty elements

The way that `XmlReader` handles empty elements presents a horrible trap. Consider the following element:

```
<customerList></customerList>
```

In XML, this is equivalent to:

```
<customerList/>
```

And yet, `XmlReader` treats the two differently. In the first case, the following code works as expected:



```
reader.ReadStartElement ("customerList");
reader.ReadEndElement();
```

In the second case, `ReadEndElement` throws an exception, because there is no separate "end element" as far as `XmlReader` is concerned. The workaround is to check for an empty element as follows:

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement ("customerList");
if (!isEmpty) reader.ReadEndElement();
```

In reality, this is a nuisance only when the element in question may contain child elements (such as a customer list). With elements that wrap simple text (such as `firstname`), you can avoid the whole issue by calling a method such as `ReadElementContentAsString`. The `ReadElement`*XXX* methods handle both kinds of empty elements correctly.

# Other ReadXXX methods

Table 11-1 summarizes all `Read`*`XXX`* methods in `XmlReader`. Most of these are designed to work with elements. The sample XML fragment shown in bold is the section read by the method described.

*Table 11-1. Read methods*

| Members | Works on NodeType | Sample XML fragment | Input parameters | Data returned |
|---|---|---|---|---|
| `ReadContentAs`*`XXX`* | `Text` | `<a>`**`x`**`</a>` | | `x` |

| | | | |
|---|---|---|---|
| ReadContentAs*XXX* | Text | <a>**x**</a> | x |

ReadString

ReadElementString

ReadElementContentAs*XXX*

ReadInnerXml

ReadOuterXml

ReadStartElement

ReadEndElement

Text

Element

Element

Element

Element
Element
Element
Element

<a>x</a>
<a>x</a>
<a>x</a>

x
x
x
x
<a>x</a>

<a>x</a>

<a>x</a>

<a>x</a>

<a>x</a>

<a>x</a>

| | | | | |
|---|---|---|---|---|
| ReadSubtree | Element | <a>x</a> | | <a>x</a> |
| ReadToDescendent | Element | <a>x</a> | "b" | |
| ReadToFollowing | Element | <a>x</a> | "b" | |
| ReadToNextSibling | Element | <a>x</a> | "b" | |
| ReadAttributeValue | Attribute | See "Reading Attributes" on page 454 | | |

The `ReadContentAs`*XXX* methods parse a text node into type *XXX*. Internally, the `XmlConvert` class performs the string-to-type conversion. The text node can be within an element or an attribute.

an element or an attribute.



The `ReadElementContentAs`*`XXX`* methods are wrappers around corresponding `Read ContentAs`*`XXX`* methods. They apply to the *element* node, rather than the *text* node enclosed by the element.

The typed `Read`*`XXX`* methods also include versions that read base 64 and BinHex formatted data into a byte array.

`ReadInnerXml` is typically applied to an element, and it reads and returns an element and all its descendents. When applied to an attribute, it returns the value of the attribute.

`ReadOuterXml` is the same as `ReadInnerXml`, except it includes rather than excludes the element at the cursor position.

`ReadSubtree` returns a proxy reader that provides a view over just the current element

`ReadSubtree` returns a proxy reader that provides a view over just the current element (and its descendents). The proxy reader must be closed before the original reader can be safely read again. At the point the proxy reader is closed, the cursor position of the original reader moves to the end of the subtree.

`ReadToDescendent` moves the cursor to the start of the first descendent node with the specified name/namespace.

`ReadToFollowing` moves the cursor to the start of the first node—regardless of depth—with the specified name/namespace.

`ReadToNextSibling` moves the cursor to the start of the first sibling node with the specified name/namespace.

`ReadString` and `ReadElementString` behave like `ReadContentAsString` and `ReadElementContentAsString`, except that they throw an exception if there's more than a *single* text node within the element. In general, these methods should be avoided, as they throw an exception if an element contains a comment.

# Reading Attributes

`XmlReader` provides an indexer giving you direct (random) access to an element's

`XmlReader` provides an indexer giving you direct (random) access to an element's attributes—by name or position. Using the indexer is equivalent to calling `GetAttribute`.

Given the following XML fragment:

```
<customer id="123" status="archived"/>
```

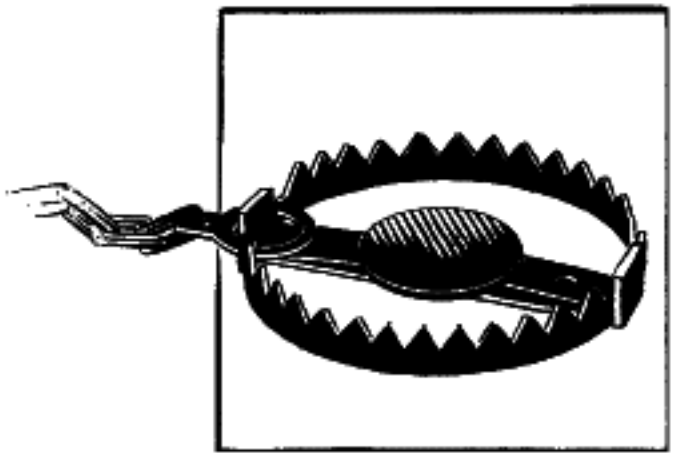
we could read its attributes as follows:

```
Console.WriteLine (reader ["id"]);
Console.WriteLine (reader ["status"]);
Console.WriteLine (reader ["bogus"] == null);
```

```
// 123
// archived
// True
```

```
// True
```





The `XmlReader` must be positioned *on a start element* in order to read attributes. *After* calling `ReadStartElement`, the attributes are gone forever!

Although attribute order is semantically irrelevant, you can access attributes by their ordinal position. We could rewrite the preceding example as follows:

```
Console.WriteLine (reader [0]);          // 123
Console.WriteLine (reader [1]);          // archived
```

The indexer also lets you specify the attribute's namespace—if it has one.

`AttributeCount` returns the number of attributes for the current node.



# Attribute nodes

To explicitly traverse attribute nodes, you must make a special diversion from the

To explicitly traverse attribute nodes, you must make a special diversion from the normal path of just calling `Read`. A good reason to do so is if you want to parse attribute values into other types, via the `ReadContentAs`*`XXX`* methods.

The diversion must begin from a *start element*. To make the job easier, the forward-only rule is relaxed during attribute traversal: you can jump to any attribute (forward or backward) by calling `MoveToAttribute`.

`MoveToElement` returns you to the `start` element from anyplace within the attribute node diversion.

Returning to our previous example:

```
<customer id="123" status="archived"/>
```

we can do this:

```
reader.MoveToAttribute ("status");
string status = ReadContentAsString();
```

```
string status = ReadContentAsString();
reader.MoveToAttribute ("id");
int id = ReadContentAsInt();
```

`MoveToAttribute` returns `false` if the specified attribute doesn't exist.

You can also traverse each attribute in sequence by calling the `MoveToFirstAttribute` and then the `MoveToNextAttribute` methods:

```
if (reader.MoveToFirstAttribute())
  do
  {
    Console.WriteLine (reader.Name + "=" + reader.Value);
  }
  while (reader.MoveToNextAttribute());
```

```
// OUTPUT:
```

```
// output:
```



```
id=123
status=archived
```

# Namespaces and Prefixes

`XmlReader` provides two parallel systems for referring to element and attribute names:

- `Name`
- `NamespaceURI` and `LocalName`

Whenever you read an element's `Name` property or call a method that accepts a single

Whenever you read an element's `Name` property or call a method that accepts a single `name` argument, you're using the first system. This works well if no namespaces or prefixes are present; otherwise, it acts in a crude and literal manner. Namespaces are ignored, and prefixes are included exactly as they were written. For example:

| Sample fragment | Name |
|---|---|
| `<customer ...>` | `customer` |
| `<customer xmlns='blah' ...>` | `customer` |
| `<x:customer ...>` | `x:customer` |

The following code works with the first two cases:

```
reader.ReadStartElement ("customer");
```

The following is required to handle the third case:

```
reader.ReadStartElement ("x:customer");
```

The second system works through two *namespace-aware* properties: `NamespaceURI`

The second system works through two *namespace-aware* properties: `NamespaceURI` and `LocalName`. These properties take into account prefixes and default namespaces defined by parent elements. Prefixes are automatically expanded. This means that `NamespaceURI` always reflects the semantically correct namespace for the current element, and `LocalName` is always free of prefixes.

When you pass two name arguments into a method such as `ReadStartElement`, you're using this same system. For example, consider the following XML:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
    ...
```

We could read this as follows:

```
reader.ReadStartElement ("customer", "DefaultNamespace");
reader.ReadStartElement ("address",  "DefaultNamespace");
reader.ReadStartElement ("city",     "OtherNamespace");
```

Abstracting away prefixes is usually exactly what you want. If necessary, you can