Chapter 11. Other XML and JSON Technologies

In Chapter 10, we covered the LINQ-to-XML API—and XML in general. In this chapter, we explore the low-level `XmlReader`/`XmlWriter` classes and the types for working with JavaScript Object Notation (JSON), which has become a popular alternative to XML.

In the online supplement, we describe the tools for working with XML schema and stylesheets.

XmlReader

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

Consider the following XML file, customer.xml:

```
<?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:

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

> **NOTE**
>
> Because `XmlReader` lets you read from potentially slow sources (`Streams` and URIs), it offers asynchronous versions of most of its methods so that you can easily write nonblocking code. We cover asynchrony in detail in Chapter 14.

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

```
using 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:

```
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 `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 `false`.

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.

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) are populated.

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);
while (reader.Read())
{
  Console.Write (new string (' ', reader.Depth * 2));  // Write indentation
  Console.Write (reader.NodeType.ToString());

  if (reader.NodeType == XmlNodeType.Element ||
      reader.NodeType == XmlNodeType.EndElement)
```

```
    {
      Console.Write (" Name=" + reader.Name);
    }
    else if (reader.NodeType == XmlNodeType.Text)
    {
      Console.Write (" Value=" + reader.Value);
    }
    Console.WriteLine ();
  }
```

The output is as follows:

```
XmlDeclaration
Element Name=customer
  Element Name=firstname
    Text Value=Jim
  EndElement Name=firstname
  Element Name=lastname
    Text Value=Bo
  EndElement Name=lastname
EndElement Name=customer
```

NOTE

Attributes are not included in `Read`-based traversal (see “Reading Attributes”).

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

`None`	`Comment`	`Document`
`XmlDeclaration`	`Entity`	`DocumentType`
`Element`	`EndEntity`	`DocumentFragment`
`EndElement`	`EntityReference`	`Notation`
`Text`	`ProcessingInstruction`	`Whitespace`
`Attribute`	`CDATA`	`SignificantWhitespace`

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.

> **NOTE**
>
> `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 `Element` 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

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

as follows:

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

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);

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();      // Skip over that pesky comment
r.ReadEndElement();     // Read the closing customer tag
```

NOTE

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 "Patterns for Using XmlReader/XmlWriter".

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 the following:

```
<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:

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

In reality, this is a nuisance only when the element in question might contain child elements (such as a customer list). With elements that wrap simple text (such as `firstname`), you can avoid the entire 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 retur
ReadContentAs*XXX*	Text	<a>**x**		x
ReadElementContentAs*XXX*	Element	**<a>x**		x
ReadInnerXml	Element	**<a>x**		x
ReadOuterXml	Element	**<a>x**		<a>x</
ReadStartElement	Element	**<a>**x		
ReadEndElement	Element	<a>x****		
ReadSubtree	Element	**<a>x**		<a>x</
ReadToDescendant	Element	**<a>x**	"b"	
ReadToFollowing	Element	**<a>x**	"b"	
ReadToNextSibling	Element	**<a>x**	"b"	
ReadAttributeValue	Attribute	See “Reading Attributes”		

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.

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

`ReadInnerXml` is typically applied to an element, and it reads and returns an element and all its descendants. When applied to an attribute, it returns the value of the attribute. `ReadOuterXml` is the same except that 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 (and its descendants). The proxy reader must be closed before the original reader can be safely read again. When the proxy reader is closed, the cursor position of the original reader moves to the end of the subtree.

`ReadToDescendant` moves the cursor to the start of the first descendant 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.

There are also two legacy methods: `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. You should avoid these methods because 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 attributes—by name or position. Using the indexer is equivalent to calling `GetAttribute`.

Given the XML fragment

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

we could read its attributes, as follows:

```
Console.WriteLine (reader ["id"]);              // 123
Console.WriteLine (reader ["status"]);          // archived
Console.WriteLine (reader ["bogus"] == null);   // True
```

> **WARNING**
>
> 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 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`.

NOTE

`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 = reader.ReadContentAsString();

reader.MoveToAttribute ("id");
int id = reader.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:
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 `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` 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 see what prefix was used through the `Prefix` property and convert it into a namespace by calling `LookupNamespace`.

XmlWriter

`XmlWriter` is a forward-only writer of an XML stream. The design of `XmlWriter` is symmetrical to `XmlReader`.

As with `XmlTextReader`, you construct an `XmlWriter` by calling `Create` with an optional `settings` object. In the following example, we enable indenting to make the output more human-readable and then write a simple XML file:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using XmlWriter writer = XmlWriter.Create ("foo.xml", settings);

writer.WriteStartElement ("customer");
writer.WriteElementString ("firstname", "Jim");
writer.WriteElementString ("lastname", "Bo");
writer.WriteEndElement();
```

This produces the following document (the same as the file we read in the first example of `XmlReader`):

```
<?xml version="1.0" encoding="utf-8"?>
<customer>
  <firstname>Jim</firstname>
```

```
  <lastname>Bo</lastname>
</customer>
```

`XmlWriter` automatically writes the declaration at the top unless you indicate otherwise in `XmlWriterSettings` by setting `OmitXmlDeclaration` to `true` or `ConformanceLevel` to `Fragment`. The latter also permits writing multiple root nodes—something that otherwise throws an exception.

The `WriteValue` method writes a single text node. It accepts both string and nonstring types such as `bool` and `DateTime`, internally calling `XmlConvert` to perform XML-compliant string conversions:

```
writer.WriteStartElement ("birthdate");
writer.WriteValue (DateTime.Now);
writer.WriteEndElement();
```

In contrast, if we call

```
WriteElementString ("birthdate", DateTime.Now.ToString());
```

the result would be both non-XML-compliant and vulnerable to incorrect parsing.

`WriteString` is equivalent to calling `WriteValue` with a string. `XmlWriter` automatically escapes characters that would otherwise be illegal within an attribute or element, such as `&`, `< >`, and extended Unicode characters.

Writing Attributes

You can write attributes immediately after writing a `start` element:

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

To write nonstring values, call `WriteStartAttribute`, `WriteValue`, and then `WriteEndAttribute`.

Writing Other Node Types

`XmlWriter` also defines the following methods for writing other kinds of nodes:

```
WriteBase64       // for binary data
WriteBinHex       // for binary data
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

`WriteRaw` directly injects a string into the output stream. There is also a `WriteNode` method that accepts an `XmlReader`, echoing everything from the given `XmlReader`.

Namespaces and Prefixes

The overloads for the `Write*` methods allow you to associate an element or attribute with a namespace. Let's rewrite the contents of the XML file in our previous example. This time we will associate all of the elements with the *http://oreilly.com* namespace, declaring the prefix `o` at the `customer` element:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");
writer.WriteElementString ("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString ("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement();
```

The output is now as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Notice how for brevity `XmlWriter` omits the child element's namespace declarations when they are already declared by the parent element.

Patterns for Using XmlReader/XmlWriter

Working with Hierarchical Data

Consider the following classes:

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>();
  public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name;                }
```

Suppose that you want to use `XmlReader` and `XmlWriter` to serialize a `Contacts` object to XML, as in the following:

```
<?xml version="1.0" encoding="utf-8"?>
<contacts>
   <customer id="1">
      <firstname>Jay</firstname>
      <lastname>Dee</lastname>
   </customer>
   <customer>                     <!-- we'll assume id is optional -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
   </customer>
   <supplier>
      <name>X Technologies Ltd</name>
   </supplier>
</contacts>
```

The best approach is not to write one big method, but to encapsulate XML functionality in the `Customer` and `Supplier` types themselves by writing

`ReadXml` and `WriteXml` methods on these types. The pattern for doing so is straightforward:

- `ReadXml` and `WriteXml` leave the reader/writer at the same depth when they exit.
- `ReadXml` reads the outer element, whereas `WriteXml` writes only its inner content.

Here's how we would write the `Customer` type:

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;

  public Customer () { }
  public Customer (XmlReader r) { ReadXml (r); }

  public void ReadXml (XmlReader r)
  {
    if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
    r.ReadStartElement();
    FirstName = r.ReadElementContentAsString ("firstname", "");
    LastName = r.ReadElementContentAsString ("lastname", "");
    r.ReadEndElement();
  }

  public void WriteXml (XmlWriter w)
  {
    if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString());
    w.WriteElementString ("firstname", FirstName);
    w.WriteElementString ("lastname", LastName);
  }
}
```

Notice that `ReadXml` reads the outer start and end element nodes. If its caller did this job instead, `Customer` couldn't read its own attributes. The reason for not making `WriteXml` symmetrical in this regard is twofold:

- The caller might need to choose how the outer element is named.

- The caller might need to write extra XML attributes, such as the element's *subtype* (which could then be used to decide which class to instantiate when reading back the element).

Another benefit of following this pattern is that it makes your implementation compatible with `IXmlSerializable` (we cover this in "Serialization" in the online supplement at *http://www.albahari.com/nutshell*).

The `Supplier` class is analogous:

```
public class Supplier
{
  public const string XmlName = "supplier";
  public string Name;

  public Supplier () { }
  public Supplier (XmlReader r) { ReadXml (r); }

  public void ReadXml (XmlReader r)
  {
    r.ReadStartElement();
    Name = r.ReadElementContentAsString ("name", "");
    r.ReadEndElement();
  }

  public void WriteXml (XmlWriter w) =>
    w.WriteElementString ("name", Name);
}
```

With the `Contacts` class, we must enumerate the `customers` element in `ReadXml`, checking whether each subelement is a customer or a supplier. We also need to code around the empty element trap:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;           // This ensures we don't get
  r.ReadStartElement();                      // snookered by an empty
  if (isEmpty) return;                       // <contacts/> element!
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add (new Customer (r));
```

```
    else if (r.Name == Supplier.XmlName) Suppliers.Add (new Supplier (r));
    else
      throw new XmlException ("Unexpected node: " + r.Name);
  }
  r.ReadEndElement();
}

public void WriteXml (XmlWriter w)
{
  foreach (Customer c in Customers)
  {
    w.WriteStartElement (Customer.XmlName);
    c.WriteXml (w);
    w.WriteEndElement();
  }
  foreach (Supplier s in Suppliers)
  {
    w.WriteStartElement (Supplier.XmlName);
    s.WriteXml (w);
    w.WriteEndElement();
  }
}
```

Here's how to serialize a `Contacts` object populated with `Customers` and `Suppliers` to an XML file:

```
var settings = new XmlWriterSettings();
settings.Indent = true;  // To make visual inspection easier

using XmlWriter writer = XmlWriter.Create ("contacts.xml", settings);

var cts = new Contacts()
// Add Customers and Suppliers...

writer.WriteStartElement ("contacts");
cts.WriteXml (writer);
writer.WriteEndElement();
```

Here's how to deserialize from the same file:

```
var settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.IgnoreComments = true;
settings.IgnoreProcessingInstructions = true;
```

```
using XmlReader reader = XmlReader.Create("contacts.xml", settings);
reader.MoveToContent();
var cts = new Contacts();
cts.ReadXml(reader);
```

Mixing XmlReader/XmlWriter with an X-DOM

You can fly in an X-DOM at any point in the XML tree where `XmlReader` or `XmlWriter` becomes too cumbersome. Using the X-DOM to handle inner elements is an excellent way to combine X-DOM's ease of use with the low-memory footprint of `XmlReader` and `XmlWriter`.

Using XmlReader with XElement

To read the current element into an X-DOM, you call `XNode.ReadFrom`, passing in the `XmlReader`. Unlike `XElement.Load`, this method is not "greedy" in that it doesn't expect to see a whole document. Instead, it reads just the end of the current subtree.

For instance, suppose that we have an XML logfile structured as follows:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

If there were a million `logentry` elements, reading the entire thing into an X-DOM would waste memory. A better solution is to traverse each `logentry` with an `XmlReader` and then use `XElement` to process the elements individually:

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

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

```
r.ReadStartElement ("log");
while (r.Name == "logentry")
{
  XElement logEntry = (XElement) XNode.ReadFrom (r);
  int id = (int) logEntry.Attribute ("id");
  DateTime date = (DateTime) logEntry.Element ("date");
  string source = (string) logEntry.Element ("source");
  ...
}
r.ReadEndElement();
```

If you follow the pattern described in the previous section, you can slot an `XElement` into a custom type's `ReadXml` or `WriteXml` method without the caller ever knowing you've cheated! For instance, we could rewrite `Customer`'s `ReadXml` method, as follows:

```
public void ReadXml (XmlReader r)
{
  XElement x = (XElement) XNode.ReadFrom (r);
  ID = (int) x.Attribute ("id");
  FirstName = (string) x.Element ("firstname");
  LastName = (string) x.Element ("lastname");
}
```

`XElement` collaborates with `XmlReader` to ensure that namespaces are kept intact, and prefixes are properly expanded—even if defined at an outer level. So, if our XML file reads like this:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
  ...
```

the `XElements` we constructed at the `logentry` level would correctly inherit the outer namespace.

Using XmlWriter with XElement

You can use an `XElement` just to write inner elements to an `XmlWriter`. The following code writes a million `logentry` elements to an XML file using `XElement`—without storing the entire thing in memory:

```
using XmlWriter w = XmlWriter.Create ("logfile.xml");

w.WriteStartElement ("log");
for (int i = 0; i < 1000000; i++)
{
  XElement e = new XElement ("logentry",
                 new XAttribute ("id", i),
                 new XElement ("date", DateTime.Today.AddDays (-1)),
                 new XElement ("source", "test"));
  e.WriteTo (w);
}
w.WriteEndElement ();
```

Using an `XElement` incurs minimal execution overhead. If we amend this example to use `XmlWriter` throughout, there's no measurable difference in execution time.

Working with JSON

JSON has become a popular alternative to XML. Although it lacks the advanced features of XML (such as namespaces, prefixes, and schemas), it benefits from being simple and uncluttered, with a format similar to what you would get from converting a JavaScript object to a string.

Historically, .NET had no built-in support for JSON, and you had to rely on third-party libraries—primarily Json.NET. Although this is no longer the case, the Json.NET library is still popular for a number of reasons:

- It's been around since 2011.
- The same API also runs on older .NET platforms.
- It's considered to be more functional (as least in the past) than the Microsoft JSON APIs.

The Microsoft JSON APIs have the advantage of having been designed from the ground up to be simple and extremely efficient. Also, from .NET 6, their functionality has become quite close to that of Json.NET.

In this section, we cover the following:

- The forward-only reader and writer (`Utf8JsonReader` and `Utf8JsonWriter`)
- The `JsonDocument` read-only DOM reader
- The `JsonNode` read/write DOM reader/writer

In "Serialization," in the online supplement at *http://www.albahari.com/nutshell*, we cover `JsonSerializer`, which automatically serializes and deserializes JSON to classes.

Utf8JsonReader

`System.Text.Json.Utf8JsonReader` is an optimized forward-only reader for UTF-8 encoded JSON text. Conceptually, it's like the `XmlReader` introduced earlier in this chapter, and is used in much the same way.

Consider the following JSON file named *people.json*:

```
{
  "FirstName":"Sara",
  "LastName":"Wells",
  "Age":35,
  "Friends":["Dylan","Ian"]
}
```

The curly braces indicate a *JSON object* (which contains *properties* such as `"FirstName"` and `"LastName"`), whereas the square brackets indicate a *JSON array* (which contains repeating elements). In this case, the repeating elements are strings, but they could be objects (or other arrays).

The following code parses the file by enumerating its JSON *tokens*. A token is the beginning or end of an object, the beginning or end of an array, the name of a property, or an array or property value (string, number, true, false, or null):

```
byte[] data = File.ReadAllBytes ("people.json");
Utf8JsonReader reader = new Utf8JsonReader (data);
while (reader.Read())
```

```
{
  switch (reader.TokenType)
  {
    case JsonTokenType.StartObject:
      Console.WriteLine ($"Start of object");
      break;
    case JsonTokenType.EndObject:
      Console.WriteLine ($"End of object");
      break;
    case JsonTokenType.StartArray:
      Console.WriteLine();
      Console.WriteLine ($"Start of array");
      break;
    case JsonTokenType.EndArray:
      Console.WriteLine ($"End of array");
      break;
    case JsonTokenType.PropertyName:
      Console.Write ($"Property: {reader.GetString()}");
      break;
    case JsonTokenType.String:
      Console.WriteLine ($" Value: {reader.GetString()}");
      break;
    case JsonTokenType.Number:
      Console.WriteLine ($" Value: {reader.GetInt32()}");
      break;
    default:
      Console.WriteLine ($"No support for {reader.TokenType}");
      break;
  }
}
```

Here’s the output:

```
Start of object
Property: FirstName Value: Sara
Property: LastName Value: Wells
Property: Age Value: 35
Property: Friends
Start of array
 Value: Dylan
 Value: Ian
End of array
End of object
```

Because Utf8JsonReader works directly with UTF-8, it steps through the tokens without first having to convert the input into UTF-16 (the format of .NET strings). Conversion to UTF-16 takes place only when you call a method such as GetString().

Interestingly, Utf8JsonReader's constructor does not accept a byte array, but rather a ReadOnlySpan<byte> (for this reason, Utf8JsonReader is defined as a *ref struct*). You can pass in a byte array because there's an implicit conversion from T[] to ReadOnlySpan<T>. In Chapter 23, we describe how spans work and how you can use them to improve performance by minimizing memory allocations.

JsonReaderOptions

By default, Utf8JsonReader requires that the JSON conform strictly to the JSON RFC 8259 standard. You can instruct the reader to be more tolerant by passing an instance of JsonReaderOptions to the Utf8JsonReader constructor. The options allow the following:

C-Style comments

> By default, comments in JSON cause a JsonException to be thrown. Setting the CommentHandling property to JsonCommentHandling.Skip causes comments to be ignored, whereas JsonCommentHandling.Allow causes the reader to recognize them and emit JsonTokenType.Comment tokens when they are encountered. Comments cannot appear in the middle of other tokens.

Trailing commas

> Per the standard, the last property of an object and the last element of an array must not have a trailing comma. Setting the AllowTrailingCommas property to e relaxes this restriction.

Control over the maximum nesting depth

> By default, objects and arrays can nest to 64 levels. Setting the `MaxDepth` to a different number overrides this setting.

Utf8JsonWriter

`System.Text.Json.Utf8JsonWriter` is a forward-only JSON writer. It supports the following types:

- `String` and `DateTime` (which is formatted as a JSON string)
- The numeric types `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, and `Decimal` (which are formatted as JSON numbers)
- `bool` (formatted as JSON true/false literals)
- JSON null
- Arrays

You can organize these data types into objects in accordance with the JSON standard. It also lets you write comments, which are not part of the JSON standard but are often supported by JSON parsers in practice.

The following code demonstrates its use:

```
var options = new JsonWriterOptions { Indented = true };

using (var stream = File.Create ("MyFile.json"))
using (var writer = new Utf8JsonWriter (stream, options))
{
  writer.WriteStartObject();
  // Property name and value specified in one call
  writer.WriteString ("FirstName", "Dylan");
  writer.WriteString ("LastName", "Lockwood");
  // Property name and value specified in separate calls
  writer.WritePropertyName ("Age");
  writer.WriteNumberValue (46);
  writer.WriteCommentValue ("This is a (non-standard) comment");
```

```
    writer.WriteEndObject();
  }
```

This generates the following output file:

```
  {
    "FirstName": "Dylan",
    "LastName": "Lockwood",
    "Age": 46
    /*This is a (non-standard) comment*/
  }
```

From .NET 6, `Utf8JsonWriter` has a `WriteRawValue` method to emit a string or byte array directly into the JSON stream. This is useful in special cases—for instance, if you want a number to be written such that it always includes a decimal point (`1.0` rather than `1`).

In this example, we set the `Indented` property on `JsonWriterOptions` to `true` to improve readability. Had we not done so, the output would be as follows:

```
  {"FirstName":"Dylan","LastName":"Lockwood","Age":46...}
```

The `JsonWriterOptions` also has an `Encoder` property to control the escaping of strings, and a `SkipValidation` property to allow structural validation checks to be bypassed (allowing the emission of invalid output JSON).

JsonDocument

`System.Text.Json.JsonDocument` parses JSON data into a read-only DOM composed of `JsonElement` instances that are generated on demand. Unlike `Utf8JsonReader`, `JsonDocument` lets you access elements randomly.

`JsonDocument` is one of two DOM-based APIs for working with JSON, the other being `JsonNode` (which we will cover in the following section).

`JsonNode` was introduced in .NET 6, primarily to satisfy the demand for a writable DOM. However, it's also suitable in read-only scenarios and exposes a somewhat more fluent interface, backed by a traditional DOM that uses classes for JSON values, arrays, and objects. In contrast, `JsonDocument` is extremely lightweight, comprising just one class of note (`JsonDocument`) and two lightweight structs (`JsonElement` and `JsonProperty`) that parse the underlying data on demand. The difference is illustrated in Figure 11-1.

> **NOTE**
>
> In most real-world scenarios, the performance benefits of `JsonDocument` over `JsonNode` are negligible, so you can skip to `JsonNode` if you prefer to learn just one API.

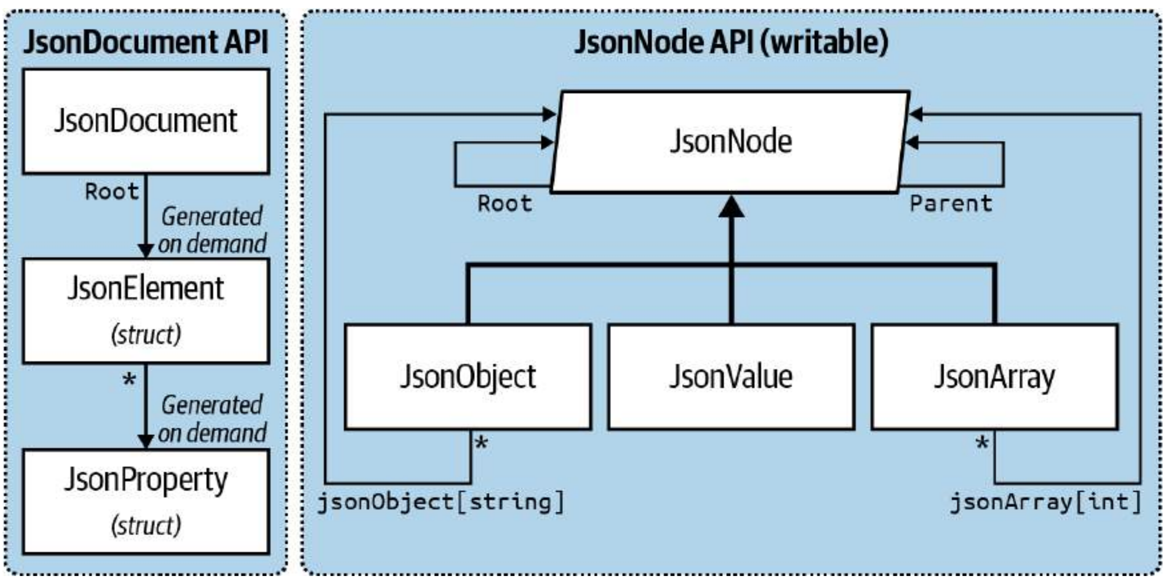

Figure 11-1. JSON DOM APIs

> **WARNING**
>
> `JsonDocument` further improves its efficiency by employing pooled memory to minimize garbage collection. This means that you must dispose the `JsonDocument` after use; otherwise, its memory will not be returned to the pool. Consequently, when a class stores a `JsonDocument` in a field, it must also implement `IDisposable`. Should this be burdensome, consider using `JsonNode` instead.

The static `Parse` method instantiates a `JsonDocument` from a stream, string, or memory buffer:

```
using JsonDocument document = JsonDocument.Parse (jsonString);
...
```

When calling `Parse`, you can optionally provide a `JsonDocumentOptions` object to control the handling of trailing commas, comments, and the maximum nesting depth (for a discussion on how these options work, see "JsonReaderOptions").

From there, you can access the DOM via the `RootElement` property:

```
using JsonDocument document = JsonDocument.Parse ("123");
JsonElement root = document.RootElement;
Console.WriteLine (root.ValueKind);       // Number
```

`JsonElement` can represent a JSON value (string, number, true/false, null), array, or object; the `ValueKind` property indicates which.

> **NOTE**
>
> The methods that we describe in the following sections throw an exception if the element isn't of the kind expected. If you're not sure of a JSON file's schema, you can avoid such exceptions by checking `ValueKind` first (or by using the `TryGet*` methods).
>
> `JsonElement` also provides two methods that work for any kind of element: `GetRawText()` returns the inner JSON, and `WriteTo` writes that element to a `Utf8JsonWriter`.

Reading simple values

If the element represents a JSON value, you can obtain its value by calling `GetString`, `GetInt32`, `GetBoolean`, etc.):

```
using JsonDocument document = JsonDocument.Parse ("123");
int number = document.RootElement.GetInt32();
```

`JsonElement` also provides methods to parse JSON strings into other commonly used CLR types such as `DateTime` (and even base-64 binary). There are also `TryGet*` versions that avoid throwing an exception if the parse fails.

Reading JSON arrays

If the `JsonElement` represents an array, you can call the following methods:

`EnumerateArray()`

Enumerates all the subitems for a JSON array (as `JsonElement`s).

`GetArrayLength()`

Returns the number of elements in the array.

You can also use the indexer to return an element at a specific position:

```
using JsonDocument document = JsonDocument.Parse (@"[1, 2, 3, 4, 5]");
int length = document.RootElement.GetArrayLength();   // 5
int value  = document.RootElement[3].GetInt32();      // 4
```

Reading JSON objects

If the element represents a JSON object, you can call the following methods:

`EnumerateObject()`

Enumerates all of the object's property names and values.

`GetProperty (string propertyName)`

Gets a property by name (returning another `JsonElement`). Throws an exception if the name isn't present.

`TryGetProperty (string propertyName, out JsonElement value)`

Returns an object's property if present.

For example:

```
using JsonDocument document = JsonDocument.Parse (@"{ ""Age"": 32}");
JsonElement root = document.RootElement;
int age = root.GetProperty ("Age").GetInt32();
```

Here's how we could "discover" the `Age` property:

```
JsonProperty ageProp = root.EnumerateObject().First();
string name = ageProp.Name;             // Age
JsonElement value = ageProp.Value;
Console.WriteLine (value.ValueKind);    // Number
Console.WriteLine (value.GetInt32());   // 32
```

JsonDocument and LINQ

`JsonDocument` lends itself well to LINQ. Given the following JSON file:

```
[
  {
    "FirstName":"Sara",
    "LastName":"Wells",
    "Age":35,
    "Friends":["Ian"]
  },
  {
    "FirstName":"Ian",
    "LastName":"Weems",
    "Age":42,
```

```
    "Friends":["Joe","Eric","Li"]
  },
  {
    "FirstName":"Dylan",
    "LastName":"Lockwood",
    "Age":46,
    "Friends":["Sara","Ian"]
  }
]
```

we can use `JsonDocument` to query this with LINQ, as follows:

```
using var stream = File.OpenRead (jsonPath);
using JsonDocument document = JsonDocument.Parse (json);

var query =
  from person in document.RootElement.EnumerateArray()
  select new
  {
    FirstName = person.GetProperty ("FirstName").GetString(),
    Age = person.GetProperty ("Age").GetInt32(),
    Friends =
      from friend in person.GetProperty ("Friends").EnumerateArray()
      select friend.GetString()
  };
```

Because LINQ queries are lazily evaluated, it's important to enumerate the query before the document goes out of scope and `JsonDocument` is implicitly disposed of by virtue of the `using` statement.

Making updates with a JSON writer

Although `JsonDocument` is read-only, you can send the content of a `JsonElement` to a `Utf8JsonWriter` with the `WriteTo` method. This provides a mechanism for emitting a modified version of the JSON. Here's how we can take the JSON from the preceding example and write it to a new JSON file that includes only people with two or more friends:

```
using var json = File.OpenRead (jsonPath);
using JsonDocument document = JsonDocument.Parse (json);

var options = new JsonWriterOptions { Indented = true };
```

```
using (var outputStream = File.Create ("NewFile.json"))
using (var writer = new Utf8JsonWriter (outputStream, options))
{
  writer.WriteStartArray();
  foreach (var person in document.RootElement.EnumerateArray())
  {
    int friendCount = person.GetProperty ("Friends").GetArrayLength();
    if (friendCount >= 2)
      person.WriteTo (writer);
  }
}
```

If you need the ability to update the DOM, however, `JsonNode` is a better solution.

JsonNode

`JsonNode` (in `System.Text.Json.Nodes`) was introduced in .NET 6, primarily to satisfy the demand for a writable DOM. However, it's also suitable in read-only scenarios and exposes a somewhat more fluent interface, backed by a traditional DOM that uses classes for JSON values, arrays, and objects (see Figure 11-1). Being classes, they incur a garbage-collection cost, but this is likely to be negligible in most real-world scenarios. `JsonNode` is still highly optimized and can actually be faster than `JsonDocument` when the same nodes are read repeatedly (because `JsonNode`, while lazy, caches the results of parsing).

The static `Parse` method creates a `JsonNode` from a stream, string, memory buffer, or `Utf8JsonReader`:

```
JsonNode node = JsonNode.Parse (jsonString);
```

When calling `Parse`, you can optionally provide a `JsonDocumentOptions` object to control the handling of trailing commas, comments, and the maximum nesting depth (for a discussion on how these options work, see "JsonReaderOptions"). Unlike `JsonDocument`, `JsonNode` does not require disposal.

> **NOTE**
>
> Calling `ToString()` on a `JsonNode` returns a human-readable (indented) JSON string. There is also a `ToJsonString()` method, which returns a compact JSON string.
>
> From .NET 8, `JsonNode` includes a static `DeepEquals` method, so you can compare two `JsonNode` objects without first expanding them into JSON strings. There is also a `DeepClone` method from .NET 8.

`Parse` returns a subtype of `JsonNode`, which will be `JsonValue`, `JsonObject`, or `JsonArray`. To avoid the clutter of a downcast, `JsonNode` provides helper methods called `AsValue()`, `AsObject()`, and `AsArray()`:

```
var node = JsonNode.Parse ("123");  // Parses to a JsonValue
int number = node.AsValue().GetValue<int>();
// Shortcut for ((JsonValue)node).GetValue<int>();
```

However, you don't usually need to call these methods, because the most commonly used members are exposed on the `JsonNode` class itself:

```
var node = JsonNode.Parse ("123");
int number = node.GetValue<int>();
// Shortcut for node.AsValue().GetValue<int>();
```

Reading simple values

We just saw that you can extract or parse a simple value by calling `GetValue` with a type parameter. To make this even easier, `JsonNode` overloads C#'s explicit cast operators, enabling the following shortcut:

```
var node = JsonNode.Parse ("123");
int number = (int) node;
```

The types for which this works comprise the standard numeric types: `char`, `bool`, `DateTime`, `DateTimeOffset`, and `Guid` (and their nullable versions), as well as `string`.

If you're not sure whether parsing will succeed, the following code is required:

```
if (node.AsValue().TryGetValue<int> (out var number))
  Console.WriteLine (number);
```

From .NET 8, calling `node.GetValueKind()` will tell you whether the node is a string, number, array, object, or true/false.

> **NOTE**
>
> Nodes that have been parsed from JSON text are internally backed by a `JsonElement` (part of the `JsonDocument` read-only JSON API). You can extract the underlying `JsonElement` as follows:
>
> ```
> JsonElement je = node.GetValue<JsonElement>();
> ```
>
> However, this doesn't work when the node is instantiated explicitly (as will be the case when we update the DOM). Such nodes are backed not by a `JsonElement` but by the actual parsed value (see "Making updates with JsonNode").

Reading JSON arrays

A `JsonNode` that represents a JSON array will be of type `JsonArray`.

`JsonArray` implements `IList<JsonNode>`, so you can enumerate over it and access the elements like you would an array or list:

```
var node = JsonNode.Parse (@"[1, 2, 3, 4, 5]");
Console.WriteLine (node.AsArray().Count);       // 5

foreach (JsonNode child in node.AsArray())
{ ... }
```

As a shortcut, you can access the indexer directly from the `JsonNode` class:

```
Console.WriteLine ((int)node[0]);   // 1
```

From .NET 8, you can also call the `GetValues<T>` method to return the data as an `IEnumerable<T>`:

```
int[] values = node.AsArray().GetValues<int>().ToArray();
```

Reading JSON objects

A `JsonNode` that represents a JSON object will be of type `JsonObject`.

`JsonObject` implements `IDictionary<string,JsonNode>`, so you can access a member via the indexer, as well as enumerating over the dictionary's key/value pairs.

And as with `JsonArray`, you can access the indexer directly from the `JsonNode` class:

```
var node = JsonNode.Parse (@"{ ""Name"":""Alice"", ""Age"": 32}");
string name = (string) node ["Name"];   // Alice
int age = (int) node ["Age"];           // 32
```

Here's how we could "discover" the `Name` and `Age` properties:

```
// Enumerate over the dictionary's key/value pairs:
foreach (KeyValuePair<string,JsonNode> keyValuePair in node.AsObject())
{
  string propertyName = keyValuePair.Key;   // "Name" (then "Age")
  JsonNode value = keyValuePair.Value;
}
```

If you're not sure whether a property has been defined, the following pattern also works:

```
if (node.AsObject().TryGetPropertyValue ("Name", out JsonNode nameNode))
{ ... }
```

Fluent traversal and LINQ

You can reach deep into a hierarchy just with indexers. For example, given the following JSON file:

```
[
  {
    "FirstName":"Sara",
    "LastName":"Wells",
    "Age":35,
    "Friends":["Ian"]
  },
  {
    "FirstName":"Ian",
    "LastName":"Weems",
    "Age":42,
    "Friends":["Joe","Eric","Li"]
  },
  {
    "FirstName":"Dylan",
    "LastName":"Lockwood",
    "Age":46,
    "Friends":["Sara","Ian"]
  }
]
```

we can extract the second person's third friend as follows:

```
string li = (string) node[1]["Friends"][2];
```

Such a file is also easy to query via LINQ:

```
JsonNode node = JsonNode.Parse (File.ReadAllText (jsonPath));

var query =
  from person in node.AsArray()
  select new
  {
    FirstName = (string) person ["FirstName"],
    Age = (int) person ["Age"],
    Friends =
      from friend in person ["Friends"].AsArray()
      select (string) friend
  };
```

Unlike `JsonDocument`, `JsonNode` is not disposable, so we don't have to worry about the potential for disposal during lazy enumeration.

Making updates with JsonNode

`JsonObject` and `JsonArray` are mutable, so you can update their content.

The easiest way to replace or add properties to a `JsonObject` is via the indexer. In the following example, we change the Color property's value from "Red" to "White" and add a new property called "Valid":

```
var node = JsonNode.Parse ("{ \"Color\": \"Red\" }");
node ["Color"] = "White";
node ["Valid"] = true;
Console.WriteLine (node.ToJsonString());  // {"Color":"White","Valid":true}
```

The second line in that example is a shortcut for the following:

```
node ["Color"] = JsonValue.Create ("White");
```

Rather than assigning the property a simple value, you can assign it a `JsonArray` or `JsonObject`. (We will demonstrate how to construct `JsonArray` and `JsonObject` instances in the following section.)

To remove a property, first cast to `JsonObject` (or call `AsObject`) and then call the `Remove` method:

```
node.AsObject().Remove ("Valid");
```

(`JsonObject` also exposes an `Add` method, which throws an exception if the property already exists.)

`JsonArray` also lets you use the indexer to replace items:

```
var node = JsonNode.Parse ("[1, 2, 3]");
node[0] = 10;
```

Calling `AsArray` exposes the `Add`/`Insert`/`Remove`/`RemoveAt` methods. In the following example, we remove the first element in the array and add one to the end:

```
var arrayNode = JsonNode.Parse ("[1, 2, 3]");
arrayNode.AsArray().RemoveAt(0);
arrayNode.AsArray().Add (4);
Console.WriteLine (arrayNode.ToJsonString());  // [2,3,4]
```

From .NET 8, you can also update a `JsonNode` by calling `ReplaceWith`:

```
var node = JsonNode.Parse ("{ \"Color\": \"Red\" }");
var color = node["Color"];
color.ReplaceWith ("Blue");
```

Constructing a JsonNode DOM programmatically

`JsonArray` and `JsonObject` have constructors that support object initialization syntax, which allows you to build an entire `JsonNode` DOM in one expression:

```
var node = new JsonArray
{
  new JsonObject {
    ["Name"] = "Tracy",
    ["Age"] = 30,
    ["Friends"] = new JsonArray ("Lisa", "Joe")
  },
  new JsonObject {
    ["Name"] = "Jordyn",
    ["Age"] = 25,
    ["Friends"] = new JsonArray ("Tracy", "Li")
  }
};
```

This evaluates to the following JSON:

```
[
  {
    "Name": "Tracy",
    "Age": 30,
    "Friends": ["Lisa", "Joe"]
  },
  {
    "Name": "Jordyn",
    "Age": 25,
    "Friends": ["Tracy","Li"]
```

    }
]