Содержание

11

Другие технологии XML

Пространство имен `System.Xml` содержит следующие пространства имен и основные классы:

`System.Xml.*`

`XmlReader` и `XmlWriter`

Высокопроизводительные однонаправленные курсоры для чтения и записи в XML-поток.

`XmlDocument`

Представляет XML-документ в DOM-модели стиля W3C (устарел).

`System.Xml.XPath`

Инфраструктура и API-интерфейс (`XPathNavigator`) для XPath — основанного на строках языка для написания запросов XML.

System.Xml.XLinq

Современная DOM-модель, ориентированная на LINQ, которая предназначена для работы с XML.

`System.Xml.XmlSchema`

Инфраструктура и API-интерфейс для схем XSD (W3C).

`System.Xml.Xsl`

Инфраструктура и API-интерфейс (`XslCompiledTransform`) для выполнения трансформаций XSLT структур XML (W3C).

`System.Xml.Serialization`

Поддерживает сериализацию классов классов в и из XML-данных (см. главу 17).

W3C — это аббревиатура, обозначающая консорциум World Wide Web Consortium, где определяются стандарты XML.

Статический класс XmlConvert, предназначенный для разбора и форматирования XML-строк, рассматривался в главе 6.

XmlReader

XmlReader — это высокопроизводительный класс для чтения XML-потока низкоуровневым однонаправленным способом.

Взгляните на следующий XML-файл:

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

Для создания экземпляра XmlReader вызывается статический метод XmlReader.Create, которому передается объект Stream, TextReader или строка URI. Например:

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

Поскольку класс XmlReader позволяет читать из потенциально медленных источников (Stream и URI), он предлагает асинхронные версии большинства своих методов, так что можно легко писать неблокирующий код. Асинхронность подробно обсуждается в главе 14.

Ниже показано, как сконструировать экземпляр XmlReader, который читает из строки:

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

Для управления настройками разбора и проверки достоверности можно также передавать объект XmlReaderSettings. В частности, следующие три свойства XmlReaderSettings полезны для пропуска избыточного содержимого:

```
bool IgnoreComments                // Пропускать узлы комментариев?
bool IgnoreProcessingInstructions // Пропускать инструкции обработки?
bool IgnoreWhitespace              // Пропускать пробельные символы?
```

В приведенном далее примере средству чтения сообщается о том, что узлы с пробельными символами, которые отвлекают внимание в типовых сценариях, выдаваться не должны:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
  ...
```

Еще одним полезным свойством XmlReaderSettings является ConformanceLevel. Его стандартное значение Document указывает средству чтения на то, что необходимо предполагать наличие допустимого XML-документа с единственным корневым узлом. Такая проблема возникает, когда нужно прочитать только внутреннюю порцию XML, содержащую несколько узлов:

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

Чтобы прочитать это без генерации исключения, потребуется установить ConformanceLevel в Fragment.

Класс XmlReaderSettings также имеет свойство по имени CloseInput, которое указывает на то, должен ли закрываться лежащий в основе поток, когда закрывается средство чтения (в XmlWriterSettings существует аналогичное свойство под названием CloseOutput). Стандартное значение для свойств CloseInput и CloseOutput равно false.

Чтение узлов

Единицами XML-потока являются *узлы XML*. Средство чтения перемещается по потоку в текстовом порядке (сначала в глубину). Свойство Depth средства чтения возвращает текущую глубину курсора.

Наиболее простой способ чтения из XmlReader предполагает вызов метода Read. Он осуществляет перемещение на следующий узел в XML-потоке подобно методу MoveNext в интерфейсе IEnumerator. Первый вызов Read устанавливает курсор на первый узел. Когда метод Read возвращает false, это означает, что курсор переместился *за* последний узел, и в данном случае экземпляр XmlReader должен быть закрыт и освобожден.

В следующем примере мы читаем каждый узел в XML-потоке, выводя по мере продвижения тип узла:

```
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));  // Вывести отступ
    Console.WriteLine (reader.NodeType);
  }
```

Ниже показан вывод:

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

Атрибуты в обход на основе Read не включаются (см. раздел “Чтение атрибутов” далее в этой главе).

Свойство NodeType имеет тип XmlNodeType, который представляет собой перечисление со следующими членами:

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

Два строковых свойства в `XmlReader` — `Name` и `Value` — предоставляют доступ к содержимому узла. В зависимости от типа узла, наполняется либо `Name`, либо `Value` (или оба):

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.DtdProcessing = DtdProcessing.Parse;    // Требуется для чтения DTD

using (XmlReader r = XmlReader.Create ("customer.xml", settings))
  while (r.Read())
  {
    Console.Write (r.NodeType.ToString().PadRight (17, '-'));
    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:
        Console.WriteLine (r.Name + " - " + r.Value); break;

      default: break;
    }
  }
```

Для демонстрации этого мы расширим наш XML-файл, включив тип документа, сущность, CDATA и комментарий:

```
<?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! -->
</customer>
```

Сущность подобна макросу; CDATA похоже на дословную строку (`@"..."`) в C#. Ниже показан результат:

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

```
EndElement-------> quote
Element----------> notes
Text-------------->    Jim Bo is a Top Customer
EndElement-------> notes
Comment----------->   That wasn't so bad!
EndElement-------> customer
```

Класс `XmlReader` автоматически распознает сущности, так что в рассмотренном примере ссылка на сущность `&tc;` расширяется в `Top Customer`.

Чтение элементов

Часто структура читаемого XML-документа уже известна. Чтобы помочь в этом отношении, класс `XmlReader` предлагает набор методов, которые выполняют чтение, *предполагая* наличие определенной структуры. Они упрощают код и одновременно с этим выполняют некоторую проверку достоверности.

Класс `XmlReader` генерирует исключение `XmlException`, если любая проверка достоверности терпит неудачу. Класс `XmlException` имеет свойства `LineNumber` и `LinePosition`, которые указывают, где произошла ошибка — регистрация этой информации в журнале очень важна в случае крупных XML-файлов!

Метод `ReadStartElement` проверяет, что текущий `NodeType` является `Element`, и затем вызывает метод `Read`. Если указано имя, то он проверяет, что оно совпадает с именем текущего элемента.

Метод `ReadEndElement` удостоверяется в том, что текущий `NodeType` — это `EndElement`, и затем вызывает метод `Read`.

Например, мы могли бы прочитать этот узел:

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

следующим образом:

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

Метод `ReadElementContentAsString` делает все описанные ранее действия за раз. Он читает начальный элемент, текстовый узел и конечный элемент, возвращая содержимое в виде строки:

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

Второй аргумент ссылается на пространство имен, которое в этом примере оставлено пустым. Доступны также типизированные версии данного метода, такие как `ReadElementContentAsInt`, разбирающие результат. Вернемся к нашему исходному XML-документу:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit>  <!-- Да, мы вставили этот комментарий! -->
</customer>
```

Его можно прочитать следующим образом:

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

using (XmlReader r = XmlReader.Create ("customer.xml", settings))
{
  r.MoveToContent();              // Пропустить XML-объявление
  r.ReadStartElement ("customer");
  string firstName    = r.ReadElementContentAsString ("firstname", "");
  string lastName     = r.ReadElementContentAsString ("lastname", "");
  decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");

  r.MoveToContent();      // Пропустить этот надоедливый комментарий
  r.ReadEndElement();     // Читать закрывающий дескриптор customer
}
```

Метод `MoveToContent` чрезвычайно удобен. Он пропускает все малоинтересное: XML-объявления, пробельные символы, комментарии и инструкции обработки. Посредством свойств `XmlReaderSettings` можно заставить средство чтения делать большинство из этого автоматически.

Необязательные элементы

Предположим в предыдущем примере, что элемент <lastname> является необязательным. Решение очень простое:

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

Случайный порядок элементов

Примеры, приводимые в этом разделе, полагаются на то, что элементы в XML-файле расположены в установленном порядке. Чтобы справиться с элементами, представленными в другом порядке, проще всего прочитать их с помещением в дерево X-DOM. Мы покажем, как это делать, в разделе "Шаблоны для использования `XmlReader/XmlWriter`" далее в главе.

Пустые элементы

Способ, которым класс `XmlReader` обрабатывает пустые элементы, таит в себе серьезную ловушку. Рассмотрим следующий элемент:

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

Вот его эквивалент в XML:

```
<customerList/>
```

Тем не менее, `XmlReader` трактует их по-разному. В первом случае приведенный ниже код работает так, как было задумано:

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

Во втором случае метод `ReadEndElement` генерирует исключение, т.к. отсутствует отдельный "конечный элемент", на который рассчитывает класс `XmlReader`. Обходной путь предусматривает добавление проверки на предмет пустых элементов, как показано ниже:

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

В действительности эта неприятность возникает, только когда рассматриваемый элемент может содержать дочерние элементы (скажем, список заказчиков). В случае элементов, которые содержат простой текст (вроде `firstname`), проблемы можно избежать путем вызова такого метода, как `ReadElementContentAsString`. Методы `ReadElement`*XXX* корректно обрабатывают оба вида пустых элементов.

Другие методы Read*XXX*

В табл. 11.1 приведена сводка по всем методам `Read`*XXX* в классе `XmlReader`. Большинство из них предназначено для работы с элементами. Выделенная полужирным часть в примере XML-фрагмента представляет собой раздел, который читается описываемым методом.

Таблица 11.1. Методы чтения

Методы	Типы узлов, на которых методы работают	Пример XML-фрагмента	Входные параметры	Возвращаемые данные
`ReadContentAs`*XXX*	`Text`	`<a>`**x**`</a>`		x
`ReadString`	`Text`	`<a>`**x**`</a>`		x
`ReadElementString`	`Element`	`<a>x</a>`		x
`ReadElementContentAs`*XXX*	`Element`	`<a>x</a>`		x
`ReadInnerXml`	`Element`	`<a>x</a>`		x
`ReadOuterXml`	`Element`	`<a>x</a>`		`<a>x</a>`
`ReadStartElement`	`Element`	**`<a>`**`x</a>`		
`ReadEndElement`	`Element`	`<a>x`**`</a>`**		
`ReadSubtree`	`Element`	`<a>x</a>`		`<a>x</a>`
`ReadToDescendant`	`Element`	**`<a>x`**`<b></b></a>`	`"b"`	
`ReadToFollowing`	`Element`	**`<a>x`**`<b></b></a>`	`"b"`	
`ReadToNextSibling`	`Element`	**`<a>x</a`**`><b></b>`	`"b"`	
`ReadAttributeValue`	`Attribute`	См. раздел “Чтение атрибутов” далее в главе		

Методы `ReadContentAs`*XXX* разбирают текстовый узел в тип *XXX*. Внутренне класс `XmlConvert` выполняет преобразование из строки в этот тип. Текстовый узел может находиться внутри элемента или атрибута.

Методы `ReadElementContentAs`*XXX* — это оболочки вокруг соответствующих методов `ReadContentAs`*XXX*. Они применяются к узлу *элемента*, а не к *текстовому* узлу, заключенному в элемент.

Типизированные методы `Read`*XXX* также имеют версии, которые читают в байтовый массив данные в форматах Base64 и BinHex.

Метод `ReadInnerXml` обычно применяется к элементу; он читает и возвращает элемент со всеми его потомками. В случае применения к атрибуту этот метод возвращает значение атрибута.

Метод `ReadOuterXml` аналогичен `ReadInnerXml` с тем лишь отличием, что он включает, а не исключает элемент в позиции курсора.

Метод `ReadSubtree` возвращает новый экземпляр `XmlReader`, который обеспечивает представление только текущего элемента (и его потомков). Чтобы исходный `XmlReader` мог безопасно продолжить чтение, этот экземпляр должен быть закрыт. В момент, когда новый экземпляр `XmlReader` закрывается, позиция курсора исходного `XmlReader` перемещается в конец поддерева.

Метод `ReadToDescendant` перемещает курсор в начало первого узла-потомка с указанным именем/пространством имен.

Метод `ReadToFollowing` перемещает курсор в начало первого узла — независимо от глубины — с указанным именем/пространством имен.

Метод `ReadToNextSibling` перемещает курсор в начало первого родственного узла с указанным именем/пространством имен.

Методы `ReadString` и `ReadElementString` ведут себя подобно `ReadContentAsString` и `ReadElementContentAsString`, но с тем отличием, что генерируют исключение, если внутри элемента обнаруживается более *одного* текстового узла. В общем случае использования этих методов следует избегать, потому что они генерируют исключение, если элемент содержит комментарий.

Чтение атрибутов

Класс `XmlReader` предоставляет индексатор, обеспечивающий прямой (произвольный) доступ к атрибутам элемента — по имени или по позиции. Применение индексатора эквивалентно вызову метода `GetAttribute`.

Имея следующий XML-фрагмент:

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

мы могли бы прочитать его атрибуты так:

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

Для того чтобы читать атрибуты, экземпляр `XmlReader` должен быть позиционирован *на начальный элемент. После* вызова метода `ReadStartElement` атрибуты исчезают навсегда!

Хотя порядок атрибутов семантически несущественен, доступ к атрибутам возможен по их ординальным позициям. Предыдущий пример можно переписать следующим образом:

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

Индексатор также позволяет указывать пространство имен атрибута, если оно имеется.

Свойство `AttributeCount` возвращает количество атрибутов для текущего узла.

Узлы атрибутов

Для явного обхода узлов атрибутов потребуется сделать специальное ответвление от нормального пути, совершаемого простым вызовом метода `Read`. Хорошим поводом поступить так является необходимость разбора значений атрибутов в другие типы с помощью методов `ReadContentAs`*XXX*.

Ответвление должно начинаться с *начального элемента*. Для упрощения работы во время обхода атрибутов правило однонаправленности ослабляется: вызывая метод `MoveToAttribute`, можно переходить к любому атрибуту (вперед или назад).

Метод `MoveToElement` возвращает начальный элемент из любого места внутри ответвления узла атрибута.

Вернувшись к предыдущему примеру:

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

можно поступить так:

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

Метод `MoveToAttribute` возвращает `false`, если указанный атрибут не существует.

Можно также выполнить обход всех атрибутов в последовательности, вызывая метод `MoveToFirstAttribute`, а затем метод `MoveToNextAttribute`:

```
if (reader.MoveToFirstAttribute())
  do
  {
    Console.WriteLine (reader.Name + "=" + reader.Value);
  }
  while (reader.MoveToNextAttribute());
// ВЫВОД:
id=123
status=archived
```

Пространства имен и префиксы

Класс `XmlReader` предлагает две параллельные системы для ссылки на имена элементов и атрибутов:

- `Name`
- `NamespaceURI` и `LocalName`

Всякий раз, когда вы читаете свойство `Name` элемента или вызываете метод, принимающий одиночный аргумент `name`, вы используете первую систему. Такой подход хорошо работает в отсутствие каких-либо пространств имен или префиксов; в противном случае это действует в грубой и буквальной манере. Пространства имен игнорируются, а префиксы включаются в точности так, как они записаны. Например:

Пример фрагмента	Значение Name
`<customer ...>`	`customer`
`<customer xmlns='blah' ...>`	`customer`
`<x:customer ...>`	`x:customer`

Приведенный ниже код работает с первыми двумя случаями:

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

Для обработки третьего случая требуется следующий код:

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

Вторая система работает через два свойства, *осведомленные о пространствах имен*: `NamespaceURI` и `LocalName`. Упомянутые свойства принимают во внимание префиксы и стандартные пространства имен, определенные родительскими элементами. Префиксы автоматически расширяются. Это означает, что свойство `NamespaceURI` всегда отражает семантически корректное пространство имен для текущего элемента, а свойство `LocalName` всегда свободно от префиксов.

При передаче двух аргументов имен в такой метод, как `ReadStartElement`, вы применяете ту же самую систему. Например, взгляните на следующий XML-фрагмент:

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

Прочитать его можно было бы так:

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

Абстрагирование от префиксов обычно является именно тем, что нужно. При необходимости посредством свойства `Prefix` можно просмотреть, какой префикс использовался, и с помощью метода `LookupNamespace` преобразовать его в пространство имен.

XmlWriter

Класс `XmlWriter` — это однонаправленное средство записи в XML-поток. Проектное решение, положенное в основу `XmlWriter`, симметрично таковому в классе `XmlReader`.

Как и `XmlTextReader`, экземпляр `XmlWriter` конструируется вызовом метода `Create`, которому передается необязательный объект настроек. В приведенном ниже примере мы разрешаем отступы, чтобы сделать вывод удобным для восприятия человеком, и затем записываем в простой XML-файл:

```
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();
}
```

В результате получается следующий документ (тот же самый, что и в файле, который мы читали в первом примере применения класса `XmlReader`):

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

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

Класс `XmlWriter` автоматически записывает объявление в начале, если только в `XmlWriterSettings` не указано обратное путем установки свойства `OmitXmlDeclaration` в `true` или свойства `ConformanceLevel` в `Fragment`. В последнем случае также разрешается запись нескольких корневых узлов — то, что иначе приводит к генерации исключения.

Метод `WriteValue` записывает одиночный текстовый узел. Он принимает строковые и нестроковые типы, такие как `bool` и `DateTime`, внутренне используя класс `XmlConvert` для выполнения совместимых с XML преобразований строк:

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

В противоположность этому, если мы вызовем:

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

то результат окажется несовместимым с XML и уязвимым к некорректному разбору. Вызов метода `WriteString` эквивалентен вызову метода `WriteValue` со строкой. Класс `XmlWriter` автоматически защищает символы, которые в противном случае были бы недопустимыми внутри атрибута либо элемента, такие как `&` , `<` `>`, и расширенные символы Unicode.

Запись атрибутов

Атрибуты можно записывать немедленно после записи начального элемента:

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

Для записи нестроковых значений вызывайте методы `WriteStartAttribute`, `WriteValue` и `WriteEndAttribute`.

Запись других типов узлов

В классе `XmlWriter` также определены следующие методы для записи других разновидностей узлов:

```
WriteBase64      // для двоичных данных
WriteBinHex      // для двоичных данных
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

Метод `WriteRaw` внедряет строку прямо в выходной поток. Имеется также метод `WriteNode`, который принимает экземпляр `XmlReader` и копирует из него все данные.

Пространства имен и префиксы

Перегруженные версии методов `Write*` позволяют ассоциировать элемент или атрибут с пространством имен. Давайте перепишем содержимое XML-файла из предыдущего примера. На этот раз мы будем связывать все элементы с пространством имен `http://oreilly.com`, объявив префикс `o` в элементе `customer`:

```
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();
```

Вывод теперь выглядит следующим образом:

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

Обратите внимание, что для краткости класс `XmlWriter` опускает объявления пространств имен в дочерних элементах, если они уже объявлены их родительским элементом.

Шаблоны для использования XmlReader/XmlWriter

Работа с иерархическими данными

Рассмотрим следующие классы:

```
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;                 }
```

Предположим, что мы хотим применить классы `XmlReader` и `XmlWriter` для сериализации объекта `Contacts` в XML, как в приведенном ниже фрагменте:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
   <customer id="1">
      <firstname>Jay</firstname>
      <lastname>Dee</lastname>
   </customer>
   <customer>           <!-- мы будем предполагать, что id необязателен -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
   </customer>
   <supplier>
      <name>X Technologies Ltd</name>
   </supplier>
</contacts>
```

Более удачный подход заключается в том, чтобы не записывать один большой метод, а инкапсулировать XML-функциональность в самих типах `Customer` и `Supplier`, реализовав для них методы `ReadXml` и `WriteXml`. Используемый шаблон довольно прост:

- когда методы `ReadXml` и `WriteXml` завершаются, они оставляют средство чтения/записи на той же глубине;
- метод `ReadXml` читает внешний элемент, тогда как метод `WriteXml` записывает только его внутреннее содержимое.

Ниже показано, как можно было бы реализовать тип `Customer`:

```
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);
  }
}
```

Обратите внимание, что метод `ReadXml` читает узлы внешнего начального и конечного элементов. Если бы эту работу делал вызывающий компонент, то класс `Customer` мог бы не читать собственные атрибуты. Причина, по которой метод `WriteXml` не сделан симметричным в этом отношении, двояка:

- вызывающий компонент может нуждаться в выборе способа именования внешнего элемента;
- вызывающему компоненту может быть необходима запись дополнительных XML-атрибутов, таких как *подтип* элемента (который затем может применяться для принятия решения о том, экземпляр какого класса создавать при чтении данного элемента).

Другое преимущество следования этому шаблону связано с тем, что ваша реализация будет совместимой с интерфейсом `IXmlSerializable` (глава 17).

Класс `Supplier` аналогичен:

```
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);
  }
}
```

В классе `Contacts` мы должны выполнять перечисление элемента `customers` в методе `ReadXml`, проверяя, является ли каждый подэлемент заказчиком или поставщиком. Также понадобится закодировать обработку пустых элементов:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;      // Это обеспечивает корректную
  r.ReadStartElement();                 // обработку пустого
  if (isEmpty) return;                  // элемента <contacts/>
  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);
                            // Непредвиденный узел
  }
  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();
  }
}
```

Смешивание `XmlReader`/`XmlWriter` с моделью X-DOM

Переключиться на модель X-DOM можно в любой точке XML-дерева, где работа с классами `XmlReader` или `XmlWriter` становится слишком громоздкой. Использование X-DOM для обработки внутренних элементов является великолепным способом комбинирования простоты применения X-DOM и низкого расхода памяти классами `XmlReader` и `XmlWriter`.

Использование XmlReader с XElement

Чтобы прочитать текущий элемент в модель X-DOM, необходимо вызвать метод XNode.ReadFrom, передав ему экземпляр XmlReader. В отличие от XElement.Load, этот метод не является "жадным" в том, что он не ожидает увидеть целый документ. Взамен метод XNode.ReadFrom читает только до конца текущего поддерева.

В качестве примера предположим, что имеется XML-файл журнала со следующей структурой:

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

При наличии миллиона элементов logentry чтение целого журнала в модель X-DOM приведет к непроизводительному расходу памяти. Более эффективное решение предусматривает обход всех элементов logentry с помощью класса XmlReader и затем использование XElement для индивидуальной обработки каждого элемента:

```
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();
}
```

Если следовать шаблону, описанному в предыдущем разделе, вы можете поместить XElement внутрь метода ReadXml или WriteXml специального типа так, что вызывающий компонент даже не обнаружит подвоха! Например, метод ReadXml класса Customer можно было бы переписать следующим образом:

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

Класс XElement взаимодействует с классом XmlReader, чтобы гарантировать, что пространства имен остались незатронутыми, а префиксы соответствующим образом расширенными — даже если они определены на внешнем уровне. Таким образом, если содержимое XML-файла выглядит, как показано ниже:

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

то экземпляры `XElement`, сконструированные на уровне `logentry`, будут корректно наследовать внешнее пространство имен.

Использование `XmlWriter` с `XElement`

Класс `XElement` можно применять только для записи внутренних элементов в `XmlWriter`. В приведенном далее коде производится запись миллиона элементов `logentry` в XML-файл с использованием класса `XElement` — без помещения всех их в память:

```
using (XmlWriter w = XmlWriter.Create ("log.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 ();
}
```

С применением класса `XElement` связаны минимальные накладные расходы во время выполнения. Если мы изменим этот пример для повсеместного использования класса `XmlWriter`, то никакой заметной разницы в скорости выполнения не будет.

XSD и проверка достоверности схемы

Содержимое отдельного XML-документа почти всегда является специфичным для предметной области, как в случае документа Microsoft Word, документа с конфигурацией приложения или веб-службы. Для каждой предметной области XML-файл соответствует определенному шаблону. Для описания схем таких шаблонов предусмотрено несколько стандартов, которые предназначены для унификации и автоматизации процедур интерпретации и проверки достоверности XML-документов. Самым широко принятым стандартом является *XSD* (*XML Schema Definition* — определение схемы XML). Его предшественники, DTD и XDR, также поддерживаются пространством имен `System.Xml`.

Взгляните на следующий XML-документ:

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

Определение XSD для этого документа можно записать так:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:int" use="required" />
            <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Как видите, сами XSD-документы представляются с помощью XML. Более того, XSD-документ может быть описан посредством XSD — вы найдете это определение по адресу `http://www.w3.org/2001/xmlschema.xsd`.

Выполнение проверки достоверности схемы

Перед чтением или обработкой файл либо документ XML можно проверить на соответствие одной или нескольким схемам. Это делается по следующим причинам:

- можно уменьшить объем проверки на предмет ошибок и обработки исключений;
- проверка достоверности схемы позволяет обнаружить ошибки, которые в противном случае остались бы незамеченными;
- сообщения об ошибках являются подробными и информативными.

Для выполнения проверки достоверности необходимо подключить схему к объекту `XmlReader`, `XmlDocument` или X-DOM и затем читать либо загружать XML-данные обычным образом. Проверка достоверности посредством схемы происходит автоматически по мере чтения содержимого, так что входной поток не читается дважды.

Проверка достоверности `XmlReader`

Ниже показано, как подключить схему из файла *`customers.xsd`* к объекту `XmlReader`:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

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

Если схема является встроенной, то вместо добавления к свойству Schemas понадобится установить следующий флаг:

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

После этого можно выполнять чтение обычным образом. Если в какой-то момент происходит отказ при проверке достоверности посредством схемы, то генерируется исключение XmlSchemaValidationException.

Вызов метода Read сам по себе обеспечивает проверку достоверности и элементов, и атрибутов: переходить к каждому отдельному атрибуту с целью его проверки не придется.

Если требуется *только* проверить документ, можно поступить так:

```
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { while (r.Read()) ; }
  catch (XmlSchemaValidationException ex)
  {
    ...
  }
```

Класс XmlSchemaValidationException имеет свойства Message, LineNumber и LinePosition. В этом случае он сообщает лишь о первой ошибке, обнаруженной в документе. Чтобы получить сведения обо всех ошибках в документе, потребуется организовать обработку события ValidationEventHandler:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  while (r.Read()) ;
```

Когда это событие обрабатывается, ошибки, связанные со схемой, больше не будут приводить к генерации исключения. Вместо этого они запускают обработчик события:

```
static void ValidationHandler (object sender, ValidationEventArgs e)
{
  Console.WriteLine ("Error: " + e.Exception.Message);
}
```

Свойство Exception класса ValidationEventArgs содержит экземпляр исключения XmlSchemaValidationException, которое сгенерировалось бы в противном случае.

В пространстве имен System.Xml также определен класс по имени XmlValidatingReader. Он предназначен для выполнения проверки достоверности схемы в версиях, предшествующих .NET Framework 2.0, и в настоящее время считается устаревшим.

Проверка достоверности X-DOM

Для выполнения проверки достоверности файла или потока XML во время его чтения в модель X-DOM необходимо создать экземпляр `XmlReader`, подключить схемы и применить средство чтения для загрузки DOM-модели:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { doc = XDocument.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }
```

С помощью расширяющих методов из пространства имен `System.Xml.Schema` можно выполнять проверку достоверности объекта `XDocument` или `XElement`, уже находящегося в памяти. Эти методы принимают экземпляр `XmlSchemaSet` (коллекция схем) и обработчик событий проверки:

```
XDocument doc = XDocument.Load (@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet ();
set.Add (null, @"customers.xsd");
StringBuilder errors = new StringBuilder ();
doc.Validate (set, (sender, args) => { errors.AppendLine
                                       (args.Exception.Message); }
              );
Console.WriteLine (errors.ToString());
```

XSLT

Аббревиатура XSLT означает *Extensible Stylesheet Language Transformations* (расширяемый язык трансформации таблиц стилей). XSLT представляет собой язык XML, который описывает преобразование одного XML-текста в другой. Наиболее типичным примером такого преобразования служит трансформация XML-документа (который обычно описывает данные) в XHTML-документ (описывающий форматированный документ).

Рассмотрим следующий XML-файл:

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

Показанный ниже XSLT-файл описывает такое преобразование:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Вывод выглядит так:

```
<html>
  <p>Jim</p>
  <p>Bo</p>
</html>
```

Класс `System.Xml.Xsl.XslCompiledTransform` эффективно выполняет XLST-преобразования. Он является заменой устаревшему классу `XmlTransform`. Класс `XmlTransform` работает очень просто:

```
XslCompiledTransform transform = new XslCompiledTransform();
transform.Load ("test.xslt");
transform.Transform ("input.xml", "output.xml");
```

Обычно удобнее пользоваться перегруженной версией метода `Transform`, которая вместо выходного файла принимает объект `XmlWriter`, что позволяет управлять форматированием.