**Использование XML**

**XML-парсеры SAX и DOM**

XML-парсер - это часть кода, которая читает XML-документ и анализирует его структуру. В этом разделе вы увидите, как работают XML-парсеры. Я покажу вам разные виды XML-парсеров и случае, когда вы можете их применять.

Здесь будет обсуждаться, как создавать парсеры и как обрабатывать результаты, которые они вам дают.

#### Как использовать парсер

Обычно парсер используется так:

1. Создаете объект парсера
2. Указываете объекту парсера на ваш XML-документ
3. Обрабатываете результаты

Очевидно, что третий шаг является сложным. Если вы знаете содержимое XML-документа, вы можете захотеть, например, генерировать Web-страницу, создать заказ на покупку или построить секторную диаграмму. Учитывая неограниченный объем данных, который может содержаться в XML-документе, задача написания приложения, которое будет корректно обрабатывать все возможные варианты входных данных, представляется ужасающей. К счастью, общие инструменты разбора XML, обсуждаемые здесь, делают эту задачу гораздо более простой.

#### Виды парсеров

Парсеры классифицируются несколькими различными способами:

* Проверяющие или непроверяющие парсеры
* Парсеры, которые поддерживают один или более языков схемы XML
* Парсеры которые поддерживают Document Object Model (DOM)
* Парсеры которые поддерживают Simple API for XML (SAX)

#### Проверяющие и непроверяющие парсеры

Есть три разных вида XML-документов:

* **Правильно форматированные документы:** Они следуют основным правилам XML (атрибуты должны быть в кавычках, теги должны быть корректно вложены и т.д.).
* **Правильные документы:** Это правильно форматированные документы, которые также удовлетворяют ряду правил, заданных в Определении Типа Документа (DTD) или в схеме XML.
* **Неправильные документы:** Любые другие.

Например, если вы имеете XML-документ, который следует всем базовым правилам XML, это *правильно форматированный* документ. Если этот документ также следует всем правилам, которые вы определили для документов расходных счетов вашей компании, он также *правильный*. Если XML-парсер находит, что XML-документ не является правильно форматированным, Спецификация XML требует, чтобы парсер выдавал фатальную ошибку. Проверка правильности, однако, - другой случай. **Проверяющие парсеры** проверяют XML-документы при их разборе, а **непроверяющие парсеры** - нет. Другими словами, если XML-документ является правильно форматированным, непроверяющий парсер не смотрит, следует ли этот документ правилам, определенным в DTD или схеме и даже вообще имеются ли для этого документа такие правила. (В большинстве проверяющих парсеров проверка правильности по умолчанию выключена.)

#### Почему используются непроверяющие парсеры?

Так почему же используются непроверяющие парсеры? So why use a non-validating парсер? Две причины: скорость и эффективность. От XML-парсер требуются некоторые затраты на чтение DTD или схемы, а затем установку движка проверки правил, который обрабатывает каждый элемент и атрибут в XML-документе в соответствии с правилами. Если вы уверены, что XML-документ правильный (Например, если он генерируется из запроса к базе данных), вы можете пропустить проверку правильности. В зависимости от сложности правил для документа, это может сэкономить значительный объем времени и памяти.

Если же у вас хрупкий код, который получает входные данные из XML-документов, и этот код требует, чтобы документы следовали определенному DTD или схеме, вам, возможно, следует проверять все, независимо от того, как дорого или долго это будет.

#### Парсеры которые поддерживают языки схемы XML

Исходная спецификация XML определила Определение Типа Документа (DTD) как способ определения того, как должен выглядеть правильный XML-документ. Большая часть синтаксиса DTD происходила из SGML и рассматривала проверку правильности с точки зрения публикации, а не с точки зрения типизации данных. Кроме того, DTD имеют синтаксис, отличный от XML-документов, что делает трудным для пользователей понимание синтаксиса правил документа.

Чтобы преодолеть эти ограничения, Worldwide Web Consortium (W3C) создал язык схемы XML. Схема XML позволяет вам определить, как выглядит правильный документ, с гораздо большей точностью. Язык схемы XML очень богат, а документы схемы XML Schema могут быть очень сложными. По этой причине XML-парсеры, которые поддерживают проверку правильности при помощи схемы XML имеют тенденцию быть очень большими. Знайте также, что язык схемы XML W3C не является единственным игроком на этом поле. Некоторые парсеры поддерживают другие языки схемы XML, такие как RELAX NG или Schematron.

#### Объектная Модель Документа (DOM)

Объектная Модель Документа или DOM является официальной рекомендацией W3C. Она определяет доступный для программ интерфейс доступа и изменения структуры XML-документов. Когда XML-парсер заявляет, что он поддерживает DOM, это означает, что он реализует интерфейсы, определенные в стандарте.

В настоящий момент два уровня DOM являются официальными рекомендациями, названные DOM Уровня 1 и DOM Уровня 2. Ожидается, что DOM Уровня 3 станет официальной рекомендацией в начале 2004. Все функции DOM, обсуждаемые в данном учебнике, являются частью DOM Уровня 1, так что кодовые примеры будут работать с любым парсером DOM.

#### Что вы получаете от парсера DOM

Когда вы разбираете XML-документ при помощи парсера DOM, вы получаете *структуру дерева*, которая представляет содержимое XML-документа. Весь текст, элементы и атрибуты (вместе с другими вещами, которые я вкратце рассмотрю) находятся в структуре дерева. DOM также предоставляет различные функции, которые вы можете использовать для исследования содержимого и структуры дерева и манипулирования ими.

Знайте, что парсер *не отслеживает* некоторых вещей. Например, количества пропусков между атрибутами и их значения не сохраняются в дереве DOM. Ч точки зрения парсеров это три атрибута совершенно одинаковы:

* type="common"
* type =  
  "common"
* type='common'

DOM не дает вам никакого способа узнать, как был кодирован исходный документ. Другой пример, - XML парсер не сообщит вам, был ли исходный элемент закодирован одним тегом или двумя. (Например, была ли горизонтальная линия XHTML закодирована как <hr/> или как <hr></hr>? ) Информация, которую должен отслеживать XML-парсер, называется **XML Infoset** .

#### Simple API for XML (SAX)

Simple API for XML (SAX) является альтернативой для работы с содержимым XML-документов. Он разработан с меньшими требованиями к объему памяти, но он перекладывает больше работы на программиста. SAX и DOM являются комплиментарными, каждый из них лучше соответствует разным ситуациям.

Являясь стандартом *де-факто*, SAX был исходно разработан David Megginson при участии многих пользователей через Internet.

#### Что вы получаете от парсера SAX

Когда вы разбираете XML-документ при помощи парсера SAX, парсер генерирует серию событий по мере того, как он читает документ. На ваше усмотрение - что делать по этим событиям. Вот несколько примеров событий, которые вы можете получить при разборе XML-документа:

* Событие startDocument.
* Для каждого элемента, событие startElement при начале элемента и событие endElement при окончании элемента.
* Если в элементе есть содержимое, будут такие события, как characters для дополнительного текста, startElement и endElement для дочерних элементов и т.п.
* Событие endDocument.

Как и в DOM, парсер SAX не показывает некоторые подробности, например, порядок появления атрибутов.

#### Как выбрать парсер

вы должны использовать парсер DOM, когда:

* Вам нужно много знать о структуре документа
* Вам нужно изменять структуру документа (может быть, вы хотите сортировать элементы, добавлять новые элементы и т.п.)
* Вам нужно обращаться к разобранной информации более одного раза

Продолжая эти широкие обобщения, скажем, что вы должны использовать парсер SAX, когда:

* У вас немного памяти (то есть, на вашей *машине* мало памяти)
* Вам нужны только несколько элементов или атрибутов из XML-документа
* Вы используете разобранную информацию только один раз

На сегодняшний день XML является одним из распространенных стандартов документов, который позволяет в удобной форме сохранять сложные по структуре данные. Поэтому разработчики платформы .NET включили в фреймворк широкие возможности для работы с XML.

Прежде чем перейти непосредственно к работе с XML-файлами, сначала рассмотрим, что представляет собой xml-документ и как он может хранить объекты, используемые в программе на c#.

Например, у нас есть следующий класс:

class User

{

    public string Name { get; set; }

    public int Age { get; set; }

    public string Company { get; set; }

}

В программе на C# мы можем создать список объектов класса User:

User user1 = new User { Name = "Bill Gates", Age = 48, Company = "Microsoft" };

User user2 = new User { Name = "Larry Page", Age = 42, Company = "Google" };

List<User> users = new List<User> { user1, user2 };

Чтобы сохранить список в формате xml мы могли бы использовать следующий xml-файл:

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

<users>

  <user name="Bill Gates">

    <company>Microsoft</company>

    <age>48</age>

  </user>

  <user name="Larry Page">

    <company>Google</company>

    <age>48</age>

  </user>

</users>

XML-документ объявляет строка <?xml version="1.0" encoding="utf-8" ?>. Она задает версию (1.0) и кодировку (utf-8) xml. Далее идет собственно содержимое документа.

XML-документ должен иметь один единственный корневой элемент, внутрь которого помещаются все остальные элементы. В данном случае таким элементом является элемент <users>. Внутри корневого элемента <users> задан набор элементов <user>. Вне корневого элемента мы не можем разместить элементы user.

Каждый элемент определяется с помощью открывающего и закрывающего тегов, например, <user> и </user>, внутри которых помещается значение или содержимое элементов. Также элемент может иметь сокращенное объявление: <user /> - в конце элемента помещается слеш.

Элемент может иметь вложенные элементы и атрибуты. В данном случае каждый элемент user имеет два вложенных элемента company и age и атрибут name.

Атрибуты определяются в теле элемента и имеют следующую форму: название="значение". Например, <user name="Bill Gates">, в данном случае атрибут называется name и имеет значение Bill Gates

Внутри простых элементов помещается их значение. Например, <company>Google</company> - элемент company имеет значение Google.

Названия элементов являются регистрозависимыми, поэтому <company> и <COMPANY> будут представлять разные элементы.

Таким образом, весь список Users из кода C# сопоставляется с корневым элементом <users>, каждый объект User - с элементом <user>, а каждое свойство объекта User - с атрибутом или вложенным элементом элемента <user>

Что использовать для свойств - вложенные элементы или атрибуты? Это вопрос предпочтений - мы можем использовать как атрибуты, так и вложенные элементы. Так, в предыдущем примере вполне можно использовать вместо атрибута вложенный элемент:

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

<users>

  <user>

    <name>Bill Gates</name>

    <company>Microsoft</company>

    <age>48</age>

  </user>

  <user>

    <name>Larry Page</name>

    <company>Google</company>

    <age>48</age>

  </user>

</users>

Теперь рассмотрим основные подходы для работы с XML, которые имеются в C#.

**Работа с XML с помощью классов System.Xml**

Для работы с XML в C# можно использовать несколько подходов. В первых версиях фреймворка основной функционал работы с XML предоставляло пространство имен **System.Xml**. В нем определен ряд классов, которые позволяют манипулировать xml-документом:

* **XmlNode**: представляет узел xml. В качестве узла может использоваться весь документ, так и отдельный элемент
* **XmlDocument**: представляет весь xml-документ
* **XmlElement**: представляет отдельный элемент. Наследуется от класса XmlNode
* **XmlAttribute**: представляет атрибут элемента
* **XmlText**: представляет значение элемента в виде текста, то есть тот текст, который находится в элементе между его открывающим и закрывающим тегами
* **XmlComment**: представляет комментарий в xml
* **XmlNodeList**: используется для работы со списком узлов

**Назначение класса XmlDocument**

Представляет XML-документ. Этот класс можно использовать для загрузки, проверки, изменения, добавления и размещения XML-кода в документе.

public class XmlDocument : System.Xml.XmlNode

Класс [XmlDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmldocument?view=netcore-3.1) — это представление XML-документа в памяти. В нем реализованы W3C по стандарту [XML модель DOM (DOM)](https://docs.microsoft.com/ru-ru/dotnet/standard/data/xml/xml-document-object-model-dom?view=netcore-3.1) уровня 1 и основной DOM уровня 2.

DOM означает объектную модель документов.

Вы можете загрузить XML в модель DOM с помощью класса [XmlDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmldocument?view=netcore-3.1), а затем программно считывать, изменять и удалять XML-документы в документе.

### Загрузка XML в объектную модель документа

Начните с XML-документа, такого как такой, который содержит несколько книг в коллекции. Он содержит основные вещи, которые можно найти в любом XML-документе, включая пространство имен, элементы, представляющие данные, и атрибуты, описывающие данные.

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

<books xmlns="http://www.contoso.com/books">

<book genre="novel" ISBN="1-861001-57-8" publicationdate="1823-01-28">

<title>Pride And Prejudice</title>

<price>24.95</price>

</book>

<book genre="novel" ISBN="1-861002-30-1" publicationdate="1985-01-01">

<title>The Handmaid's Tale</title>

<price>29.95</price>

</book>

<book genre="novel" ISBN="1-861001-45-3" publicationdate="1811-01-01">

<title>Sense and Sensibility</title>

<price>19.95</price>

</book>

</books>

Затем загрузите эти данные в модель DOM, чтобы вы могли работать с ними в памяти. Самый популярный способ сделать это — это ссылка на файл на локальном компьютере или в сети.

В этом примере загружается XML из файла. Если файл не существует, он просто создает некоторый XML-код и загружает его.

XmlDocument doc = new XmlDocument();

doc.PreserveWhitespace = true;

try { doc.Load("booksData.xml"); }

catch (System.IO.FileNotFoundException)

{

doc.LoadXml("<?xml version=\"1.0\"?> \n" +

"<books xmlns=\"http://www.contoso.com/books\"> \n" +

" <book genre=\"novel\" ISBN=\"1-861001-57-8\" publicationdate=\"1823-01-28\"> \n" +

" <title>Pride And Prejudice</title> \n" +

" <price>24.95</price> \n" +

" </book> \n" +

" <book genre=\"novel\" ISBN=\"1-861002-30-1\" publicationdate=\"1985-01-01\"> \n" +

" <title>The Handmaid's Tale</title> \n" +

" <price>29.95</price> \n" +

" </book> \n" +

"</books>");

}

**Сравнение возможностей SAX и класса XmlTextReader**

public class XmlTextReader : System.Xml.XmlReader, System.Xml.IXmlLineInfo, System.Xml.IXmlNamespaceResolver

[XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1) предоставляет доступ только для чтения к потоку XML-данных. Текущим является узел, на котором расположен модуль чтения. Модуль чтения расширен с помощью любого из методов Read и Properties, отражая значение текущего узла.

Похож на SAX. Одно из различий заключается в том, что SAX является моделью типа рассылки (push), т.е. посылает данные приложению и разработчик должен быть готов принять их, a применяет модель запроса (pull), где данные посылаются приложению, которое их запрашивает. Это предоставляет более простую и интуитивно понятную модель для программирования. Другое преимущество состоит в том, что модель запроса может быть избирательной в отношении данных, посылаемых приложению. Если нужны не все данные, то их не нужно обрабатывать. В модели рассылки все данные XML должны быть обработаны приложением, нужны они ему или нет.

Этот класс реализует [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader?view=netcore-3.1) и соответствует рекомендациям консорциума W3C язык XML (XML) 1,0 и пространству имен в XML. XmlTextReader предоставляет следующие функциональные возможности:

* Применяет правила XML правильного формата.
* XmlTextReader не обеспечивает проверку данных.
* Проверяет, что DocumentType узлы имеют правильный формат. XmlTextReader проверяет DTD на правильность формата, но не проверяет его с помощью DTD.
* Для узлов, где [NodeType](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.nodetype?view=netcore-3.1) XmlNodeType.EntityReference, возвращается один пустой узел EntityReference (то есть свойство [Value](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.value?view=netcore-3.1) имеет значение String.Empty).
* Не расширяет атрибуты по умолчанию.

Так как XmlTextReader не выполняет дополнительные проверки, необходимые для проверки данных, он предоставляет средство синтаксического анализа правильного формата.

Чтобы выполнить проверку данных, используйте проверку [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader?view=netcore-3.1).

Для чтения XML-данных из [XmlDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmldocument?view=netcore-3.1) используйте [XmlNodeReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlnodereader?view=netcore-3.1).

XmlTextReader создает [XmlException](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlexception?view=netcore-3.1) при ошибках синтаксического анализа XML. После возникновения исключения состояние модуля чтения не может быть предсказуемым. Например, тип сообщаемого узла может отличаться от фактического типа узла текущего узла. Используйте свойство [ReadState](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.readstate?view=netcore-3.1), чтобы проверить, находится ли модуль чтения в состоянии ошибки.

### Вопросы безопасности

При использовании класса [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1) необходимо учитывать следующие моменты.

* Исключения, вызванные [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1), могут раскрывать сведения о пути, которые не должны быть собраны в приложение. Приложения должны перехватывать исключения и обрабатывать их соответствующим образом.
* Обработка определения DTD по умолчанию включена. Отключите обработку DTD, если возникли проблемы отказа в обслуживании или если вы работаете с ненадежными источниками. Задайте для свойства [DtdProcessing](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.dtdprocessing?view=netcore-3.1) значение [Prohibit](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.dtdprocessing?view=netcore-3.1#System_Xml_DtdProcessing_Prohibit), чтобы отключить обработку DTD.

Если обработка DTD включена, то с помощью класса [XmlSecureResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlsecureresolver?view=netcore-3.1) можно ограничить ресурсы, доступные для объекта [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1). Кроме того, можно создать собственное приложение, ограничивающее объем памяти и время, выделяемое на обработку XML. Например, можно ограничить время ожидания в приложении ASP.NET.

* XML-данные могут содержать ссылки на внешние ресурсы, такие как DTD-файл. По умолчанию внешние ресурсы разрешаются с помощью объекта [XmlUrlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlurlresolver?view=netcore-3.1) без пользовательских учетных данных. Это можно сделать еще более безопасным, выполнив одно из следующих действий.
  + Ограничить ресурсы, к которым имеет доступ [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1), установив в качестве значения свойства [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.xmlresolver?view=netcore-3.1) объект [XmlSecureResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlsecureresolver?view=netcore-3.1).
  + Не разрешать [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader?view=netcore-3.1) открывать какие-либо внешние ресурсы, установив для свойства [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.xmlresolver?view=netcore-3.1) значение null.
* XML-данные могут содержать большое количество атрибутов, деклараций пространств имен, вложенных элементов и так далее, на обработку которых требуется значительное время. Чтобы ограничить размер входных данных, отправляемых в [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1), создайте пользовательскую реализацию IStream и укажите ее [XmlTextReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader?view=netcore-3.1).
* Метод [ReadValueChunk](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader.readvaluechunk?view=netcore-3.1) можно использовать для управления большими потоками данных. Этот метод за раз считывает небольшое число символов, а не выделяет одну строку для всего значения.
* По умолчанию общие сущности не развертываются. Общие сущности раскрываются при вызове метода [ResolveEntity](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextreader.resolveentity?view=netcore-3.1).

**Возможности класса XmlTextWriter**

w.WriteStartElement("root");

w.WriteAttributeString("xmlns", "x", null, "urn:1");

w.WriteStartElement("item","urn:1");

w.WriteEndElement();

w.WriteStartElement("item","urn:1");

w.WriteEndElement();

w.WriteEndElement();

Приведенный C# выше код выдает следующие выходные данные. XmlTextWriter продвигает объявление пространства имен к корневому элементу, чтобы избежать его дублирования в двух дочерних элементах. Дочерние элементы получают префикс из объявления пространства имен.

<root xmlns:x="urn:1">

<x:item/>

<x:item/>

</x:root>

XmlTextWriter также позволяет переопределить текущее объявление пространства имен. В следующем примере URI-код пространства имен "123" переопределяется "ABC" для создания элемента XML <x:node xmlns:x="abc"/>.

w.WriteStartElement("x","node","123");

w.WriteAttributeString("xmlns","x",null,"abc");

Используя методы Write, которые принимают префикс в качестве аргумента, можно также указать используемый префикс. В следующем примере два разных префикса сопоставляются с одним и тем же URI-кодом пространства имен для создания <x:root xmlns:x="urn:1"><y:item xmlns:y="urn:1"/></x:root>текста XML.

XmlTextWriter w = new XmlTextWriter(Console.Out);

w.WriteStartElement("x","root","urn:1");

w.WriteStartElement("y","item","urn:1");

w.WriteEndElement();

w.WriteEndElement();

w.Close();

Если несколько объявлений пространств имен сопоставляют разные префиксы с одним URI-адресом пространства имен, XmlTextWriter просматривает стек объявлений пространств имен назад и выбирает ближайший из них.

XmlTextWriter w = new XmlTextWriter(Console.Out);

w.Formatting = Formatting.Indented;

w.WriteStartElement("x","root","urn:1");

w.WriteStartElement("y","item","urn:1");

w.WriteAttributeString("attr","urn:1","123");

w.WriteEndElement();

w.WriteEndElement();

w.Close();

В приведенном C# выше примере, поскольку в вызове WriteAttributeString не указан префикс, модуль записи использует последний префикс, помещенный в стек пространства имен, и создает следующий код XML:

<x:root xmlns:x="urn:1">

<y:item y:attr="123" xmlns:y="urn:1" />

</x:root>

Если возникает конфликт пространств имен, XmlTextWriter разрешает их путем создания альтернативных префиксов. Например, если атрибут и элемент имеют один и тот же префикс, но разные пространства имен, XmlWriter создает альтернативный префикс для атрибута. Созданные префиксы имеют имя n{i} где i — число, начинающееся с 1. Для каждого элемента число сбрасывается до 1.

Атрибуты, связанные с URI пространства имен, должны иметь префикс (пространства имен по умолчанию не применяются к атрибутам). Это соответствует разделу 5,2 рекомендаций по пространствам имен W3C в XML. Если атрибут ссылается на URI пространства имен, но не указывает префикс, то модуль записи создает префикс для атрибута.

При записи пустого элемента добавляется дополнительное пространство между именем тега и закрывающим тегом, например <item />. Это обеспечивает совместимость с более старыми браузерами.

Если в качестве параметра метода используется String, null и String.Empty эквивалентны. String.Empty следуют правилам W3C.

Для записи строго типизированных данных используйте класс [XmlConvert](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlconvert?view=netcore-3.1) для преобразования типов данных в строку. Например, следующий C# код преобразует данные из Double в String и записывает <price>19.95</price>элемента.

Double price = 19.95;

writer.WriteElementString("price", XmlConvert.ToString(price));

XmlTextWriter не проверяет наличие следующих флажков:

* Недопустимые символы в именах атрибутов и элементов.
* Символы Юникода, не соответствующие заданной кодировке. Если символы Юникода не соответствуют указанной кодировке, XmlTextWriter не помещает символы Юникода в символьные сущности.
* Повторяющиеся атрибуты.
* Символы в открытом идентификаторе DOCTYPE или системном идентификаторе.

### Вопросы безопасности

При работе с классом [XmlTextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter?view=netcore-3.1) необходимо учитывать следующие моменты.

* Исключения, вызываемые [XmlTextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter?view=netcore-3.1), могут раскрывать сведения о пути, которые не должны быть собраны в приложение. Приложения должны перехватывать исключения и обрабатывать их соответствующим образом.
* При передаче [XmlTextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter?view=netcore-3.1) другому приложению ему предоставляется базовый поток. Если нужно передать [XmlTextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter?view=netcore-3.1) приложению с частичным доверием, то следует использовать объект [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter?view=netcore-3.1), созданный методом [Create](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter.create?view=netcore-3.1).
* [XmlTextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter?view=netcore-3.1) не проверяет данные, передаваемые методам [WriteDocType](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter.writedoctype?view=netcore-3.1) или [WriteRaw](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmltextwriter.writeraw?view=netcore-3.1). В эти методы не следует передавать произвольные данные.
* Если параметры по умолчанию изменены, то не гарантируется, что создаваемые выходные данные будут правильно сформированы XML-данными.
* Не принимайте вспомогательные компоненты, такие как объект [Encoding](https://docs.microsoft.com/ru-ru/dotnet/api/system.text.encoding?view=netcore-3.1), из ненадежного источника.

**Возможности класса XPath**

**XPath** представляет язык запросов в XML. Он позволяет выбирать элементы, соответствующие определенному селектору.

Рассмотрим некоторые наиболее распространенные селекторы:

.-выбор текущего узла

..-выбор родительского узла

\*-выбор всех дочерних узлов текущего узла

User - выбор всех узлов с определенным именем, в данном случае с именем "user"

@name-выбор атрибута текущего узла, после знака @ указывается название атрибута (в данном случае "name")

@+ - выбор всех атрибутов текущего узла

element[3]-выбор определенного дочернего узла по индексу, в данном случае третьего узла

//user -выбор в документе всех узлов с именем "user"

user[@name='Bill Gates']

выбор элементов с определенным значением атрибута. В данном случае выбираются все элементы "user" с атрибутом name='Bill Gates'

user[company='Microsoft']

выбор элементов с определенным значением вложенного элемента. В данном случае выбираются все элементы "user", у которых дочерний элемент "company" имеет значение 'Microsoft'

//user/company

выбор в документе всех узлов с именем "company", которые находятся в элементах "user"

Действие запросов XPath основано на применении двух методов класса **XmlElement**:

* **SelectSingleNode()**: выбор единственного узла из выборки. Если выборка по запросу содержит несколько узлов, то выбирается первый
* **SelectNodes()**: выборка по запросу коллекции узлов в виде объекта XmlNodeList

Для запросов возьмем xml-документ:

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

<users>

  <user name="Bill Gates">

    <company>Microsoft</company>

    <age>48</age>

  </user>

  <user name="Larry Page">

    <company>Google</company>

    <age>42</age>

  </user>

</users>

Теперь выберем все узлы корневого элемента, то есть все элементы user:

XmlDocument xDoc = new XmlDocument();

xDoc.Load("[D://users.xml](file:///D:\users.xml)");

XmlElement xRoot = xDoc.DocumentElement;

// выбор всех дочерних узлов

XmlNodeList childnodes = xRoot.SelectNodes("\*");

foreach (XmlNode n in childnodes)

    Console.WriteLine(n.OuterXml);

Выберем все узлы <user>:

XmlNodeList childnodes = xRoot.SelectNodes("user");

Выведем на консоль значения атрибутов name у элементов user:

XmlNodeList childnodes = xRoot.SelectNodes("user");

foreach (XmlNode n in childnodes)

  Console.WriteLine(n.SelectSingleNode("@name").Value);

Результатом выполнения будет следующий вывод:

Bill Gates

Larry Page

Выберем узел, у которого атрибут name имеет значение "Bill Gates":

XmlNode childnode = xRoot.SelectSingleNode("user[@name='Bill Gates']");

if (childnode != null)

    Console.WriteLine(childnode.OuterXml);

Выберем узел, у которого вложенный элемент "company" имеет значение "Microsoft":

XmlNode childnode = xRoot.SelectSingleNode("user[company='Microsoft']");

if (childnode != null)

    Console.WriteLine(childnode.OuterXml);

Допустим, нам надо получить только компании. Для этого надо осуществить выборку вниз по иерархии элементов:

XmlNodeList childnodes = xRoot.SelectNodes("//user/company");

foreach (XmlNode n in childnodes)

    Console.WriteLine(n.InnerText);

Ключевым классом, который позволяет манипулировать содержимым xml, является **XmlNode**, поэтому рассмотрим некоторые его основные методы и свойства:

* Свойство **Attributes** возвращает объект XmlAttributeCollection, который представляет коллекцию атрибутов
* Свойство **ChildNodes** возвращает коллекцию дочерних узлов для данного узла
* Свойство **HasChildNodes** возвращает true, если текущий узел имеет дочерние узлы
* Свойство **FirstChild** возвращает первый дочерний узел
* Свойство **LastChild** возвращает последний дочерний узел
* Свойство **InnerText** возвращает текстовое значение узла
* Свойство **InnerXml** возвращает всю внутреннюю разметку xml узла
* Свойство **Name** возвращает название узла. Например, <user> - значение свойства Name равно "user"
* Свойство **ParentNode** возвращает родительский узел у текущего узла

Применим эти классы и их функционал. И вначале для работы с xml создадим новый файл. Назовем его *users.xml* и определим в нем следующее содержание:

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

<users>

  <user name="Bill Gates">

    <company>Microsoft</company>

    <age>48</age>

  </user>

  <user name="Larry Page">

    <company>Google</company>

    <age>42</age>

  </user>

</users>

Теперь пройдемся по этому документу и выведем его данные на консоль:

using System.Xml;

class Program

{

    static void Main(string[] args)

    {

        XmlDocument xDoc = new XmlDocument();

        xDoc.Load("[D://users.xml](file:///D:\users.xml)");

        // получим корневой элемент

        XmlElement xRoot = xDoc.DocumentElement;

        // обход всех узлов в корневом элементе

        foreach(XmlNode xnode in xRoot)

        {

            // получаем атрибут name

            if(xnode.Attributes.Count>0)

            {

                XmlNode attr = xnode.Attributes.GetNamedItem("name");

                if (attr!=null)

                    Console.WriteLine(attr.Value);

            }

            // обходим все дочерние узлы элемента user

            foreach(XmlNode childnode in xnode.ChildNodes)

            {

                // если узел - company

                if(childnode.Name=="company")

                {

                    Console.WriteLine($"Компания: {childnode.InnerText}");

                }

                // если узел age

                if (childnode.Name == "age")

                {

                    Console.WriteLine($"Возраст: {childnode.InnerText}");

                }

            }

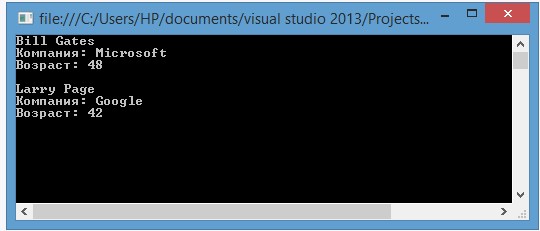
            Console.WriteLine();

        }

        Console.Read();

    }

}



Чтобы начать работу с документом xml, нам надо создать объект XmlDocument и затем загрузить в него xml-файл: xDoc.Load("users.xml");

При разборе xml для начала мы получаем корневой элемент документа с помощью свойства xDoc.DocumentElement. Далее уже происходит собственно разбор узлов документа.

В цикле foreach(XmlNode xnode in xRoot) пробегаемся по всем дочерним узлам корневого элемента. Так как дочерние узлы представляют элементы <user>, то мы можем получить их атрибуты: XmlNode attr = xnode.Attributes.GetNamedItem("name"); и вложенные элементы: foreach(XmlNode childnode in xnode.ChildNodes)

Чтобы определить, что за узел перед нами, мы можем сравнить его название: if(childnode.Name=="company")

Подобным образом мы можем создать объекты User по данным из xml:

using System;

using System.Collections.Generic;

using System.Xml;

namespace HelloApp

{

    class User

    {

        public string Name { get; set; }

        public int Age { get; set; }

        public string Company { get; set; }

    }

    class Program

    {

        static void Main(string[] args)

        {

            List<User> users = new List<User>();

            XmlDocument xDoc = new XmlDocument();

            xDoc.Load("[D://users.xml](file:///D:\users.xml)");

            XmlElement xRoot = xDoc.DocumentElement;

            foreach (XmlElement xnode in xRoot)

            {

                User user = new User();

                XmlNode attr = xnode.Attributes.GetNamedItem("name");

                if (attr != null)

                    user.Name = attr.Value;

                foreach (XmlNode childnode in xnode.ChildNodes)

                {

                    if (childnode.Name == "company")

                        user.Company = childnode.InnerText;

                    if (childnode.Name == "age")

                        user.Age = Int32.Parse(childnode.InnerText);

                }

                users.Add(user);

            }

            foreach (User u in users)

                Console.WriteLine($"{u.Name} ({u.Company}) - {u.Age}");

            Console.Read();

        }

    }

}

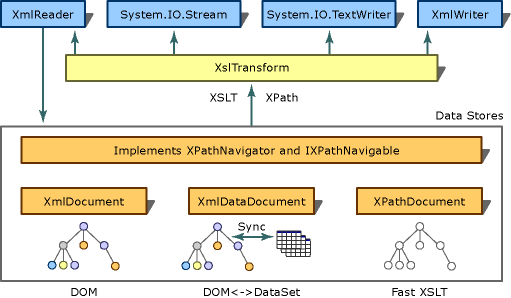
**Что такое XSLT**

**XSLT** (*e****X****tensible****S****tylesheet****L****anguage****T****ransformations*) — язык преобразования [XML](https://ru.wikipedia.org/wiki/XML)-документов. Спецификация XSLT входит в состав [XSL](https://ru.wikipedia.org/wiki/XSL) и является рекомендацией [W3C](https://ru.wikipedia.org/wiki/W3C).

При применении *таблицы стилей* XSLT, состоящей из набора *шаблонов*, к XML-документу (*исходное дерево*) образуется *конечное дерево*, которое может быть [сериализовано](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) в виде XML-документа, [XHTML](https://ru.wikipedia.org/wiki/XHTML)-документа (только для XSLT 2.0), [HTML](https://ru.wikipedia.org/wiki/HTML)-документа или [простого текстового файла](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5). Правила выбора (и, отчасти, преобразования) данных из исходного дерева пишутся на языке запросов [XPath](https://ru.wikipedia.org/wiki/XPath).

XSLT имеет множество различных применений, в основном в области [веб-программирования](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и генерации отчётов. Одной из задач, решаемых языком XSLT, является отделение данных от их представления, как часть общей парадигмы MVC ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) [*Model-view-controller*](https://ru.wikipedia.org/wiki/Model-view-controller)). Другой стандартной задачей является преобразование XML-документов из одной [XML-схемы](https://ru.wikipedia.org/wiki/XML_Schema) в другую.

Таблицы XSLT применяются для преобразования содержимого исходного XML-документа в другой документ, отличный по формату или структуре (например, чтобы преобразовать XML в HTML для использования в веб-узле или преобразовать его в документ, содержащий только поля, необходимые приложению). Это преобразование описано в [рекомендациях XSLT версии 1.0](https://www.w3.org/TR/1999/REC-xslt-19991116) консорциума W3C. В платформе .NET Framework класс [XslTransform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xsltransform), находящийся в пространстве имен [System.Xml.Xsl](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl), является обработчиком XSLT, который реализует функциональность этой спецификации. На следующем рисунке показана архитектура преобразования в платформе .NET Framework.



В рекомендациях XSLT используется язык XPath для выбора частей XML-документа, XPath - язык запросов, используемый для навигации по узлам дерева документов. Как показано на схеме, реализация языка XPath платформы .NET Framework используется для выбора частей XML-документа, сохраненных в нескольких классах, таких как [XmlDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmldocument), [XmlDataDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmldatadocument) и [XPathDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.xpathdocument). [XPathDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.xpathdocument)

**Способы применения преобразования XSLT**

# XSLT-преобразования с помощью класса XslCompiledTransform

Класс [XslCompiledTransform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform) является обработчиком XSLT в платформе Microsoft .NET Framework. Этот класс используется для компиляции таблиц стилей и выполнения преобразований XSLT.

Хотя класс [**XslCompiledTransform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform) имеет более высокий общий уровень производительности, чем класс [**XslTransform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xsltransform), метод [**Load**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.load) класса [**XslCompiledTransform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform) может выполняться медленнее, чем метод [**Load**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xsltransform.load) класса [**XslTransform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xsltransform) при первом вызове преобразования. Причина этого заключается в необходимости компиляции XSLT-файла перед его загрузкой.

# Входные данные для класса XslCompiledTransform

Метод [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) принимает три входных типа для документа источника: объект, реализующий интерфейс [IXPathNavigable](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.ixpathnavigable), объект [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader), который считывает документ источника, и строку URI.

Класс [**XslCompiledTransform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform) сохраняет пробелы по умолчанию. Это соответствует разделу [**3.4 рекомендации W3C XSLT 1.0**](https://www.w3.org/TR/xslt.html#strip).

Интерфейс [IXPathNavigable](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.ixpathnavigable) реализован в классах [XmlNode](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlnode) и [XPathDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.xpathdocument). Эти классы представляют хранящийся в памяти кэш XML-данных.

* Класс [XmlNode](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlnode) основан на модели W3C DOM и позволяет вносить изменения.
* Класс [XPathDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.xpathdocument) представляет собой доступное только для чтения хранилище данных на основе модели данных XPath. Класс [XPathDocument](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xpath.xpathdocument) рекомендуется для обработки XSLT. Он обеспечивает более высокую производительность по сравнению с классом [XmlNode](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlnode).

Преобразования применяются к документу в целом. Иными словами, если передать узел, отличный от корневого узла документа, это не помешает процессу преобразования обратиться ко всем узлам загружаемого документа. Чтобы преобразовать фрагмент узла, необходимо создать объект, содержащий только фрагмент этого узла, и передать его методу [**Transform**](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform).

В следующем примере метод [XslCompiledTransform.Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) используется для преобразования файла books.xml в books.html с помощью таблицы стилей transform.xsl.

// Open books.xml as an XPathDocument.

XPathDocument doc = new XPathDocument("books.xml");

// Create a writer for writing the transformed file.

XmlWriter writer = XmlWriter.Create("books.html");

// Create and load the transform with script execution enabled.

XslCompiledTransform transform = new XslCompiledTransform();

XsltSettings settings = new XsltSettings();

settings.EnableScript = true;

transform.Load("transform.xsl", settings, null);

// Execute the transformation.

transform.Transform(doc, writer);

Метод [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) загружает из текущего узла объекта [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader) всех его потомков. Это позволяет использовать фрагмент документа в качестве контекстного документа. После возвращения метода [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) объект [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader) позиционируется на следующем узле за контекстным документом. По достижении конца документа объект [XmlReader](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlreader) позиционируется в конец файла (EOF).

В следующем примере метод [XslCompiledTransform.Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) используется для преобразования файла books.xml в books.html с помощью таблицы стилей transform.xsl.

// Create a reader to read books.xml

XmlReader reader = XmlReader.Create("books.xml");

// Create a writer for writing the transformed file.

XmlWriter writer = XmlWriter.Create("books.html");

// Create and load the transform with script execution enabled.

XslCompiledTransform transform = new XslCompiledTransform();

XsltSettings settings = new XsltSettings();

settings.EnableScript = true;

transform.Load("transform.xsl", settings, null);

// Execute the transformation.

transform.Transform(reader, writer);

Можно также указать URI исходного документа в виде входных данных XSLT. Для разрешения URI используется объект [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver). Чтобы использовать объект [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver), нужно передать его методу [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform). Если объект [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) не указан, метод [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) использует объект [XmlUrlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlurlresolver) по умолчанию без учетных данных.

В следующем примере метод [XslCompiledTransform.Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) используется для преобразования файла books.xml в books.html с помощью таблицы стилей transform.xsl.

// Create and load the transform with script execution enabled.

XslCompiledTransform transform = new XslCompiledTransform();

XsltSettings settings = new XsltSettings();

settings.EnableScript = true;

transform.Load("transform.xsl", settings, null);

// Execute the transformation.

transform.Transform("books.xml", "books.html");

# Параметры вывода в классе XslCompiledTransform

Рассматриваются доступные параметры вывода XSLT. Параметры вывода можно указать в таблице стилей или методе [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform).

Элемент xsl:output указывает параметры вывода. Тип вывода, указанный методом [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform), определяет поведение параметров xsl:output.

В следующей таблице описано поведение каждого атрибута, доступного в элементе xsl:output, если тип вывода - поток или объект [TextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.textwriter).

| **Имя атрибута** | **Поведение** |
| --- | --- |
| method | Поддерживается. |
| version | Не обрабатывается. Версия всегда 1.0 для XML и 4.0 для HTML. |
| encoding | Не учитывается при выводе в объект [TextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.textwriter). Вместо него используется свойство [TextWriter.Encoding](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.textwriter.encoding). |
| omit-xml-declaration | Поддерживается. |
| Автономный | Поддерживается. |
| doctype-public | Поддерживается. |
| doctype-system | Поддерживается. |
| cdata-section-elements | Поддерживается. |
| indent | Поддерживается. |
| media-type | Поддерживается. |

#### Отправка выходных данных в XmlWriter

Если в таблице стилей используется элемент xsl:output, а тип вывода - объект [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter), нужно использовать свойство [XslCompiledTransform.OutputSettings](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.outputsettings) при создании объекта [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter). Свойство [XslCompiledTransform.OutputSettings](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.outputsettings) возвращает объект [XmlWriterSettings](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwritersettings), который содержит сведения, полученные из элемента xsl:output скомпилированной таблицы стилей. Этот объект [XmlWriterSettings](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwritersettings) можно передать в метод [XmlWriter.Create](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter.create), чтобы создать объект [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter) с правильными настройками.

## Типы вывода

В следующем списке описаны типы вывода, доступные в команде [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform).

#### XmlWriter

Класс [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter) записывает потоки или файлы XML. С помощью класса [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter) можно указать возможности, поддерживающие объект [XmlWriterSettings](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwritersettings), включая параметры вывода. Класс [XmlWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlwriter) представляет собой неотъемлемую часть платформы [System.Xml](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml). Используйте этот тип вывода для передачи выходных результатов в другой процесс XML.

#### Строка

Используйте этот тип вывода, чтобы указать URI выходного файла.

#### Поток

Поток - это абстракция последовательности байтов, например файла, устройства ввода-вывода, межпроцессного канала связи или сокета TCP/IP. Класс [Stream](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.stream) и его производные классы обеспечивают универсальное представление различных типов ввода и вывода, изолируя программиста от конкретных особенностей операционной системы и базовых устройств.

Используйте этот тип вывода для пересылки данных в объекты [FileStream](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.filestream), [MemoryStream](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.memorystream) или выходной поток (Response.OutputStream).

#### TextWriter

Объект [TextWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.textwriter) записывает последовательные символы. Он реализован в классах [StringWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.stringwriter) и [StreamWriter](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.streamwriter), которые записывают символы в строки и потоки соответственно. Используйте этот тип вывода, если нужно вывести строку.

## Примечания

* При записи пустых тегов между последним символом имени элемента и обратной косой чертой записывается пробел, например <myElement />. Это позволяет правильно отобразить сформированные HTML-страницы в старых браузерах.

Во время XSLT-преобразования может понадобиться несколько раз разрешать внешние ресурсы.

## Использование класса XmlResolver

Класс [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) используется для разрешения внешних ресурсов. В следующей таблице описывается, когда класс [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) участвует в XSLT-обработке.

| **Задача XSLT** | **Зачем используется класс XmlResolver** |
| --- | --- |
| Компиляция таблицы стилей | Разрешение URI таблицы стилей.  - и -  Разрешение URI-ссылок в любых элементах xsl:import или xsl:include. |
| Выполнение таблицы стилей. | Разрешение URI документа контекста.  - и -  Разрешение URI-ссылок в любых функциях XSLT document(). |

Методы [Load](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.load) и [Transform](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.transform) включают в себя перегружаемые, принимающие объект [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) в качестве одного из аргументов. Если [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) не указан, используется [XmlUrlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlurlresolver) по умолчанию без учетных данных.

Следующий список описывает, когда может потребоваться указать объект [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver):

* Если процессу XSLT требуется доступ к сетевому ресурсу, требующему проверки подлинности, можно использовать [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) с необходимыми учетными данными.
* Если необходимо ограничить ресурсы, к которым имеет доступ процесс XSLT, можно использовать [XmlSecureResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlsecureresolver) с надлежащим набором разрешений. Использовать класс [XmlSecureResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlsecureresolver) рекомендуется, если вам необходимо открыть ресурс, которым вы не управляете или к которому нет доверия.
* Если необходимо настроить поведение особым образом, можно реализовать собственный класс [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver) и использовать его для разрешения ресурсов.
* Если необходимо убедиться в отсутствии доступа к внешним ресурсам, можно указать null в качестве значения аргумента [XmlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlresolver).

В следующем примере компилируется таблица стилей, которая хранится на сетевом ресурсе. Объект [XmlUrlResolver](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xmlurlresolver) задает учетные данные, необходимые для доступа к таблице стилей.

// Create the credentials.

NetworkCredential myCred = new NetworkCredential(UserName,SecurelyStoredPassword,Domain);

CredentialCache myCache = new CredentialCache();

myCache.Add(new Uri("http://www.contoso.com/"), "Basic", myCred);

myCache.Add(new Uri("http://app.contoso.com/"), "Basic", myCred);

// Set the credentials on the XmlUrlResolver object.

XmlUrlResolver resolver = new XmlUrlResolver();

resolver.Credentials = myCache;

// Compile the style sheet.

XslCompiledTransform xslt = new XslCompiledTransform();

xslt.Load("http://serverName/data/xsl/order.xsl",XsltSettings.Default, resolver);

С помощью класса [XsltArgumentList](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xsltargumentlist) можно добавлять объекты расширения или параметры. Затем объекты расширения или параметры можно вызывать из таблицы стилей. Кроме того, можно внедрять в таблицу стилей блоки скрипта с помощью элемента msxsl:script.

# Выполнение XSLT-преобразования с помощью сборки

XSLT-компилятор (xsltc.exe) компилирует таблицы стилей XSLT и создает сборку. Скомпилированную таблицу стилей можно передать непосредственно методу [XslCompiledTransform.Load(Type)](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.load#System_Xml_Xsl_XslCompiledTransform_Load_System_Type_). Подписанные сборки не могут создаваться с помощью xsltc.exe.

Средство xsltc.exe входит в состав среды Visual Studio.

Синтаксис

xsltc [options] [/class:<name>] <sourceFile> [[/class:<name>] <sourceFile>...]

| **Аргумент** | **Описание** |
| --- | --- |
| sourceFile | Задает имя таблицы стилей. Таблица стилей должна быть локальным файлом или располагаться в интрасети. |
| Параметр | Описание |
| /c[lass]: name | Задает имя класса для следующей таблицы стилей. Имя класса может быть полным.  По умолчанию для имени класса используется имя таблицы стилей. Например, если компилируется таблица стилей customers.xsl, по умолчанию используется имя класса customers. |
| /debug[+|-] | Указывает, создаются ли отладочные сведения.  Задание значения + или /debug включает создание компилятором отладочных сведений, которые размещаются в файле базы данных программы (PDB). Создаваемый PDB-файл получает имя assemblyName.pdb.  Если задано значение -, которое действует, если не указан параметр /debug, то отладочные данные не создаются. Создается конечная версия сборки. Примечание. Компиляция в режиме отладки может значительно снизить производительность XSLT. |
| /help | Отображает синтаксис команд и параметров программы. |
| /nologo | Отключает вывод сообщения об авторских правах для компилятора. |
| /platform: string | Задает платформы, на которых может работать сборка. Далее описаны допустимые значения платформ:  x86 - сборка компилируется для работы в 32-разрядной среде CLR, совместимой с архитектурой x86;  x64 - сборка компилируется для работы в 64-разрядной среде CLR на компьютере, поддерживающем набор инструкций AMD64 или EM64T;  Itanium компилирует сборку для выполнения в 64-разрядной среде CLR на компьютере с процессором Itanium.  anycpu - сборка компилируется для работы на любой платформе Это значение по умолчанию. |
| /out: assemblyName | Задает имя выводимой сборки. По умолчанию имя сборки совпадает с именем главной таблицы стилей или первой таблицы стилей, если присутствует несколько таблиц стилей.  Если таблица стилей содержит скрипты, они сохраняются в отдельной сборке. Имена сборок скриптов формируются из имени главной сборки. Например, если для имени сборки указано CustOrders.dll, первая сборка скрипта получит имя CustOrders\_Script1.dll. |
| /settings: document+-, script+-, DTD+-, | Указывает, допускаются ли в таблице стилей функции document(), скрипт XSLT или DTD.  По умолчанию поддержка DTD, функции document() и скриптов отключена. |
| @ file | Позволяет указать файл, содержащий параметры компилятора. |
| ? | Отображает синтаксис команд и параметров программы. |

Решения XSLT могут состоять из нескольких модулей таблиц стилей. Программа xsltc.exe создает сборки из таблиц стилей. Затем эти сборки можно передать методу [XslCompiledTransform.Load(Type)](https://docs.microsoft.com/ru-ru/dotnet/api/system.xml.xsl.xslcompiledtransform.load#System_Xml_Xsl_XslCompiledTransform_Load_System_Type_). Это может снизить затраты производительности в некоторых сценариях развертывания XSLT.

Необходимо также включить в приложение скомпилированную сборку в виде ссылки.

Программа xsltc.exe не проверяет имена класса (/class:name) и сборки (/out:assemblyName). Если имена оказываются недопустимыми, среда CLR сообщает об ошибке.

Примеры

Следующая команда компилирует таблицу стилей и создает сборку с именем booksort.dll

xsltc booksort.xsl

Следующая команда компилирует таблицу стилей и создает сборку и PDB-файл, которые называются booksort.dll и booksort.pdb соответственно.

xsltc booksort.xsl /debug

Следующая команда компилирует таблицу стилей, содержащую элемент msxsl:script, и создает две сборки с именами calc.dll и calc\_Script1.dll.

xsltc /settings:script+ calc.xsl

Следующая команда включает обработку DTD и поддержку скриптов, а затем создает две сборки с именами myTest.dll и myTest\_Script1.dll.

xsltc /settings:DTD+,script+ /out:myTest calc.xsl

Следующая команда компилирует два модуля таблицы стилей и создает одну сборку с именем booksort.dll

xsltc booksort.xsl output.xsl

### Копирование XML-файлов и XSLT-файлов на локальный компьютер

Скопируйте XSLT-файл на локальный компьютер и переименуйте его в Transform.xsl.

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:msxsl="urn:schemas-microsoft-com:xslt"

xmlns:user="urn:my-scripts">

<msxsl:script language="C#" implements-prefix="user">

<![CDATA[

public string discount(string price){

char[] trimChars = { '$' };

//trim leading $, convert price to type double

double discount\_value = Convert.ToDouble(price.TrimStart(trimChars));

//apply 10% discount and round appropriately

discount\_value = .9\*discount\_value;

//convert value to decimal and format as currency

string discount\_price = discount\_value.ToString("C");

return discount\_price;

}

]]>

</msxsl:script>

<xsl:template match="catalog">

<html>

<head></head>

<body>

<table border="1">

<tr>

<th align="left">Title</th>

<th align="left">Author</th>

<th align="left">Genre</th>

<th align="left">Publish Date</th>

<th align="left">Price</th>

</tr>

<xsl:for-each select="book">

<tr>

<td>

<xsl:value-of select="title"/>

</td>

<td>

<xsl:value-of select="author"/>

</td>

<td>

<xsl:value-of select="genre"/>

</td>

<td>

<xsl:value-of select="publish\_date"/>

</td>

<xsl:choose>

<xsl:when test="genre = 'Fantasy'">

<td>

<xsl:value-of select="user:discount(price)"/>

</td>

</xsl:when>

<xsl:otherwise>

<td>

<xsl:value-of select="price"/>

</td>

</xsl:otherwise>

</xsl:choose>

</tr>

</xsl:for-each>

</table>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

Скопируйте XML-файл на локальный компьютер и переименуйте его в books.xml.

<?xml version="1.0"?>

<catalog>

<book id="bk101">

<author>Gambardella, Matthew</author>

<title>XML Developer's Guide</title>

<genre>Computer</genre>

<price>$44.95</price>

<publish\_date>2000-10-01</publish\_date>

</book>

<book id="bk102">

<author>Ralls, Kim</author>

<title>Midnight Rain</title>

<genre>Fantasy</genre>

<price>$5.95</price>

<publish\_date>2000-12-16</publish\_date>

</book>

<book id="bk103">

<author>Corets, Eva</author>

<title>Maeve Ascendant</title>

<genre>Fantasy</genre>

<price>$5.95</price>

<publish\_date>2000-11-17</publish\_date>

</book>

<book id="bk106">

<author>Randall, Cynthia</author>

<title>Lover Birds</title>

<genre>Romance</genre>

<price>$4.95</price>

<publish\_date>2000-09-02</publish\_date>

</book>

<book id="bk107">

<author>Thurman, Paula</author>

<title>Splish Splash</title>

<genre>Romance</genre>

<price>$4.95</price>

<publish\_date>2000-11-02</publish\_date>

</book>

</catalog>

### Компиляция таблицы стилей с поддержкой скриптов

При выполнении следующей команды из командной строки создаются две сборки с именами Transform.dll и Transform\_Script1.dll (это действие выполняется по умолчанию. Если не указано иное, имя класса и сборки по умолчанию совпадает с именем главной таблицы стилей):

xsltc /settings:script+ Transform.xsl

Следующая команда явно задает имя класса Transform:

xsltc /settings:script+ /class:Transform Transform.xsl

### Включение скомпилированной сборки в виде ссылки во время компиляции кода

1. Можно включить сборку в Visual Studio, добавив ссылку в обозревателе решений, или из командной строки.
2. Если используется язык C#, введите в командной строке следующее:

csc myCode.cs /r:system.dll;system.xml.dll;Transform.dll

### Использование скомпилированной сборки в коде

В следующем примере показано, как выполнить преобразование XSLT, используя скомпилированную таблицу стилей.

using System;

using System.Xml.Xsl;

class Example

{

static void Main()

{

//Create a new XslCompiledTransform and load the compiled transformation.

XslCompiledTransform xslt = new XslCompiledTransform();

xslt.Load(typeof(Transform));

// Execute the transformation and output the results to a file.

xslt.Transform("books.xml", "discount\_books.html");

}

}

Чтобы создать динамическую ссылку на скомпилированную сборку, замените

xslt.Load(typeof(Transform));

xslt.Load(System.Reflection.Assembly.Load("Transform").GetType("Transform"));

**Применения Linq для XML**

## Преимущества LINQ to XML

### Конструирование деревьев XML было упрощено с помощью функционального конструирования

Код создания дерева XML крайне громоздок, при использовании стандартного API-интерфейса для работы с XML. После создания документа XML понадобится создать некоторого типа узел XML, такой как элемент, установить его значение и добавить к родительскому элементу. Однако каждый из этих трех шагов должен быть выполнен отдельно с использованием W3C DOM API. Это приводит к неясной схеме и объемному коду. В этом API-интерфейсе не предусмотрена поддержка создания элемента или любого типа узла в определенном месте дерева XML по отношению к его родителю и его инициализации за одну операцию.

**API-интерфейс LINQ to XML** не только предоставляет ту же возможность создания дерева XML, что и W3C DOM API, но также предлагает новую технику, называемую функциональным конструированием, для создания дерева XML. ***Функциональное конструирование*** позволяет схеме диктовать то, как конструируются объекты XML и инициализируются их значения, и все это — одновременно, в единственном операторе. API-интерфейс достигает этого за счет предоставления конструкторов новых XML-объектов, которые принимают в качестве параметров как отдельные объекты, так и их множества, указывая их значения. Тип добавляемого объекта или объектов определяет то, где именно в схеме они располагаются. Общий шаблон выглядит следующим образом:

*XMLObject оbj =*

*new XMLObject (ObjectName,*

*XMLObject1,*

*XMLObject2,*

*...*

*XMLObjectN);*

Приведенный фрагмент является просто псевдокодом, предназначенным для иллюстрации шаблона. Ни один из классов, присутствующих в нем, не существует на самом деле; они лишь представляют некоторый концептуальный абстрактный класс XML. При добавлении к элементу, реализованному классом XElement, атрибута XML, который реализован классом XAttribute из LINQ to XML, этот атрибут становится атрибутом данного элемента. Например, если в предыдущем псевдокоде XMLObject1 добавляется к вновь созданному XMLObject по имени оbj, и оbj является XElement, a XMLObject1 — XAttribute, то XMLObject1 становится атрибутом XElement по имени оbj.

Если XElement добавляется к XElement, то добавляемый XElement становится дочерним элементом того, к которому он добавлен. Поэтому, например, если XMLObject1 и оbj являются элементами, то XMLObject1 становится дочерним элементом оbj.

При создании экземпляра объекта XMLObject, как показано в приведенном псевдокоде, можно задавать его содержимое, указывая от 1 до N объектов XMLObject. Как будет показано в разделе "Создание текста с помощью XText" в статье "[Создание XML](https://professorweb.ru/my/LINQ/linq_xml/level5/5_3.php)", можно даже указывать его содержимое с помощью строки, поскольку строка автоматически преобразуется в XMLObject.

Все это совершенно логично и составляет суть функционального конструирования. Ниже приведен пример:

using System.Xml.Linq;

...

XElement employees =

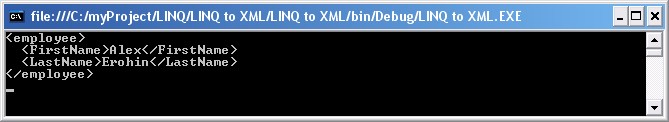
new XElement("employee",

new XElement("FirstName","Alex"),

new XElement("LastName","Erohin"));

Console.WriteLine(employees.ToString());

Обратите внимание, что при конструировании элемента по имени employees в качестве его значения передаются два объекта XElement, каждый из которых становится его дочерним элементом. Также обратите внимание, что при конструировании элементов FirstName и LastName вместо указания дочерних объектов, указаны просто текстовые значения элементов. Вот результат работы этого кода:



Обратите внимание, насколько проще инициализировать в коде схему XML. Код LINQ to XML существенно короче, чем при использовании XML DOM. Для примера сравните следующие два листинга кода, выполняющих одинаковую работу, но использующие два разных интерфейса:

**// Используем старый API-интерфейс XML DOM**

// Объявляю некоторые переменные, которые будут использоваться повторно.

XmlElement xmlEmployee;//Employee

XmlAttribute xmlEmployeeType; //EmployeeType

XmlElement xmlFirstName;

XmlElement xmlLastName;

// Сначала я должен построить документ XML.

XmlDocument xmlDoc = new XmlDocument();

// Создаю корневой элемент и добавляю его в документ.

XmlElement xmlEmployees = xmlDoc.CreateElement("Employees");

xmlDoc.AppendChild(xmlEmployees);

// Создаю список участников

xmlEmployee = xmlDoc.CreateElement("Employee");

xmlEmployeeType = xmlDoc.CreateAttribute("type");

xmlEmployeeType.InnerText = "Programmer";

xmlEmployee.Attributes.Append(xmlEmployeeType);

xmlFirstName = xmlDoc.CreateElement("FirstName");

xmlFirstName.InnerText = "Alex";

xmlEmployee.AppendChild(xmlFirstName);

xmlLastName = xmlDoc.CreateElement("LastName");

xmlLastName.InnerText = "Erohin";

xmlEmployee.AppendChild(xmlLastName);

xmlEmployees.AppendChild(xmlEmployee);

// Создаю еще одного участника

xmlEmployee = xmlDoc.CreateElement("Employee");

xmlEmployeeType = xmlDoc.CreateAttribute("type");

xmlEmployeeType.InnerText = "Editor";

xmlEmployee.Attributes.Append(xmlEmployeeType);

xmlFirstName = xmlDoc.CreateElement("FirstName");

xmlFirstName.InnerText = "Elena";

xmlEmployee.AppendChild(xmlFirstName);

xmlLastName = xmlDoc.CreateElement("LastName");

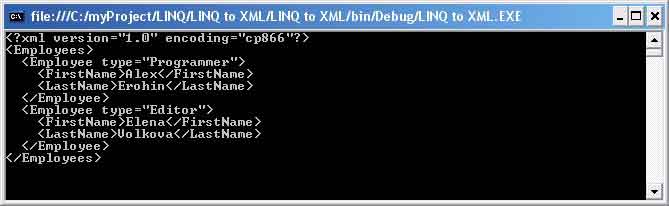
xmlLastName.InnerText = "Volkova";

xmlEmployee.AppendChild(xmlLastName);

xmlEmployees.AppendChild(xmlEmployee);

// Выводим на консоль

xmlDoc.Save(Console.Out);



Просто взглянув на него, невозможно получить никакого представления о том, как должна выглядеть результирующая структура XML.

Давайте теперь создадим то же самое дерево XML используя LINQ to XML:

**// Используем новый API-интерфейс LINQ to XML**

XElement xEmployees =

new XElement("Employees",

new XElement("Employee",

new XAttribute("type", "Programmer"),

new XElement("FirstName", "Alex"),

new XElement("LastName", "Erohin")),

new XElement("Employee",

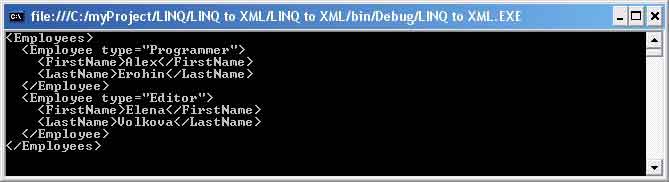
new XAttribute("type", "Editor"),

new XElement("FirstName", "Elena"),

new XElement("LastName", "Volkova")));

Console.WriteLine(xEmployees);

Результат получается аналогичным предыдущему, за исключением того, что мы не создаем новый XML-документ, поэтому в начале нет строки с кодировкой и версией XML:



Как видите, здесь пришлось создать намного меньше кода, что впоследствии упростит сопровождение. К тому же схему намного легче понять, просто прочитав этот код.

### Центральная роль элемента вместо документа

В исходном W3C DOM API нельзя просто создать XML-элемент — XmlElement; нужно было иметь XML-документ — XmlDocument, из которого создавать его. Попытка создать экземпляр XmlElement вроде следующего:

XmlElement xmlEmployee = new XmlElement ("Employee");

привела бы к ошибке компиляции:

*'System.Xml.XmlElement.XmlElement (string, string, string, System.Xml.XmlDocument)'*

*is inaccessible due to its protection level*

*'System.Xml.XmlElement.XmlElement (string, string, string, System.Xml.XmlDocument)'*

*не доступен из-за своего уровня защиты*

С помощью W3C DOM API создавать XmlElement можно только вызовом метода XmlDocument по имени CreateElement, как показано ниже:

XmlDocument xmlDoc = new XmlDocument();

XmlElement xmlEmployee = xmlDoc.CreateElement ("Employee");

Этот код компилируется успешно. Но часто обязательное создание документа XML причиняет неудобства, особенно когда нужно просто создать элемент XML. Новый LINQ-оснащенный XML API позволяет создавать экземпляр элемента без создания документа XML:

XElement xEmployee = new XElement("Employee");

XML-элементы - не единственный тип узлов, подчиняющихся этому ограничению W3C DOM. Атрибуты, комментарии, разделы CData, инструкции обработки и ссылки на сущности — все это должно было создаваться из документа XML. К счастью LINQ to XML дает возможность непосредственно создавать каждый из этих объектов "на лету".

Конечно, ничто не мешает создавать XML-документы с помощью нового API-интерфейса. Например:

XDocument xDocument =

new XDocument(

new XElement("Employees",

new XElement("Employee",

new XAttribute("type", "Programmer"),

new XElement("FirstName", "Alex"),

new XElement("LastName", "Erohin")),

new XElement("Employee",

new XAttribute("type", "Editor"),

new XElement("FirstName", "Elena"),

new XElement("LastName", "Volkova"))));

Console.WriteLine(xDocument.ToString());

Разметка XML, произведенная предыдущим кодом, идентична XML-разметке, которая была создана ранее.

В дополнение к созданию деревьев XML без документов XML можно делать большинство тех же вещей, которые требуют документов, вроде чтения XML-разметки из файла и сохранение ее в файл.

### Имена, пространства имен и префиксы

Чтобы исключить путаницу, связанную с именами, пространствами имен и префиксами пространств имен, последние изъяты из API-интерфейса. С помощью LINQ to XML префиксы пространств имен разворачиваются на вводе и возвращаются в выводе. Внутри они не существуют!

Пространство имен используется в XML с целью уникальной идентификации схемы XML для некоторой части дерева XML. В качестве пространств имен XML используются URI, потому что они уже уникальны для каждой организации. В некоторых примерах кода дерево XML создается следующим образом:

<Employees>

<Employee type="Programmer">

<FirstName>Alex</FirstName>

<LastName>Erohin</LastName>

</Employee>

<Employee type="Editor">

<FirstName>Elena</FirstName>

<LastName>Volkova</LastName>

</Employee>

</Employees>

Любой код, обрабатывающий эти данные XML, будет написан в предположении, что узел Employees может содержать множество узлов Employee, каждый из которых имеет атрибут type и узлы FirstName и LastName. Но что, если коду также придется обрабатывать XML из другого источника, в котором окажется узел Employees, но с отличающейся внутренней схемой?

Пространство имен известит код о том, как должна выглядеть схема, позволяя ему соответствующим образом обработать XML. В XML каждый элемент должен иметь имя. Когда элемент создается, если его имя указано в конструкторе, оно неявно преобразуется из string в **объект XName**. Объект XName состоит из пространства имен — объекта XNamespace — и своего локального имени, того, которое было указано. Поэтому, например, можно создать элемент Employees следующим образом:

XElement xEmployees = new XElement("Employees");

При создании элемента объект XName получает пустое пространство имен и локальное имя Employees. Если во время отладки этой строки кода просмотреть переменную xEmployees в окне слежения, то увидите, что ее член Name установлен в {Employees}. Развернув член Name, вы увидите, что он содержит член LocalName, установленный в Employees, и член по имени Namespace, который будет пустым — {}. В данном случае пространство имен отсутствует.

Чтобы указать пространство имен, понадобится просто создать объект XNamespace и предварить им локальное имя следующим образом:

XNamespace nameSpace = "http://www.anyurl.ru";

XElement xEmployees = new XElement(nameSpace + "Employees");

После этого при просмотре элемента xEmployees в окне слежения отладчика можно увидеть, что Name установлено в http://www.anyurl.ru/Employees. Разворачивание члена Name покажет, что LocalName будет по-прежнему Employees, а член Namespace установлен в http://www.anyurl.ru.

Для того чтобы указать пространство имен, не обязательно использовать объект XNamespace. Пространство имен можно задать в виде жестко закодированного строкового литерала:

XElement xEmployees = new XElement("{http://www.anyurl.ru}" + "Employees");

Обратите внимание, что пространство имен заключено в фигурные скобки. Это указывает конструктору XElement на тот факт, что данная часть означает пространство имен. Если вновь просмотреть член Name объекта Employees в окне слежения, можно увидеть, что член Name и встроенные в него члены LocalName и Namespace установлены идентично значениям, установленным в предыдущем примере, где для создания элемента применялся объект XNamespace.

Имейте в виду, что при установке пространства имен простого указания URI компании или домена организации может быть недостаточно для того, чтобы гарантировать уникальность. Это гарантирует лишь отсутствие коллизий с любой другой существующей организацией, которая соблюдает установленные соглашения названий пространств имен. Однако внутри организации могут случиться коллизии с любым другим подразделением, если в пространстве имен не будет указано ничего помимо URI организации. И здесь весьма пригодится знание организационной структуры предприятия — его подразделений, департаментов и т.д. Лучше всего, если пространство имен будет раскрывать весь путь до определенного уровня, находящегося под вашим контролем.

XNamespace nameSpace = "http://www.anyurl.ru/LINQ";

XDocument xDocument =

new XDocument(

new XElement("Employees",

// Добавляем пространство имен с префиксом

new XAttribute(XNamespace.Xmlns + "linq", nameSpace),

new XElement("Employee",

new XAttribute("type", "Programmer"),

new XElement("FirstName", "Alex"),

new XElement("LastName", "Erohin")),

new XElement("Employee",

new XAttribute("type", "Editor"),

new XElement("FirstName", "Elena"),

new XElement("LastName", "Volkova"))));

Console.WriteLine(xDocument.ToString());

В приведенном коде в качестве префикса пространства имен указывается linq и с помощью объекта XAttribute в схему включается спецификация префикса.

### Извлечение значения узла

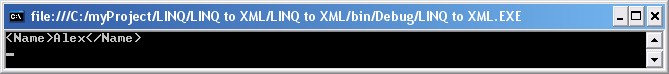
Если вы ознакомились с примером использования старого интерфейса XML DOM из предыдущего раздела, то наверняка вспомните, что вам приходилось сталкиваться с той же проблемой, которая помешала сразу получить результат, т.е. действительное значение узла. Если нет достаточного опыта работы с кодом XML DOM, то неизбежно возникает ошибка подобного рода, поскольку многие забывают о необходимости выполнения дополнительного шага для получения значения узла.

В LINQ to XML эта проблема очень изящно решена. Метод ToString элемента выводит саму строку XML, а не тип объекта, как это делается в W3C DOM API. Это очень удобно, когда необходим определенный фрагмент XML из некоторой точки дерева, к тому же имеет намного больше смысла, чем вывод типа объекта. Ниже приведен пример:

XElement name = new XElement("Name", "Alex");

Console.WriteLine(name.ToString());

Нажатие <Ctrl+F5> выдаст следующий результат:



Уже лучше. Однако можно добиться еще лучшего результата. Разумеется, дочерние узлы включены в вывод, и поскольку метод WriteLine не имеет явной перегрузки, принимающей XElement, он вызывает метод ToString. Что еще более важно — если привести узел к типу данных, к которому может быть преобразовано его значение, то будет выведено само значение. Ниже показан пример:

XElement name = new XElement("Person",

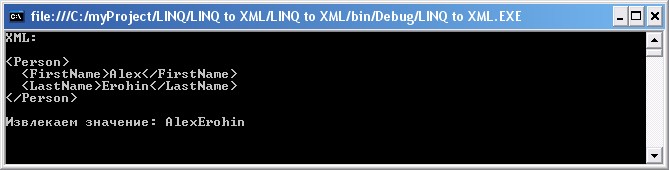
new XElement("FirstName","Alex"),

new XElement("LastName","Erohin"));

Console.WriteLine("XML: \n\n" + name +

"\n\nИзвлекаем значение: " + (string)name);

Вот результат этого кода:



В то время как все эти примеры упрощают получение значения элемента, все они представляют собой случаи приведения элемента к тому типу данных, который изначально имело значение этого элемента. Это не обязательно. Все, что нужно — чтобы значение элемента могло быть преобразовано к указанному типу данных.

К сожалению то, как именно значения преобразуются, не задано, но на самом деле для этой цели используются методы класса [System.Xml.XmlConvert](https://msdn.microsoft.com/en-us/library/system.xml.xmlconvert.aspx). Код из примера ниже демонстрирует, как это происходит при приведении к типу bool:

try

{

XElement smoker = new XElement("Smoker", "Tue");

Console.WriteLine(smoker);

Console.WriteLine((bool)smoker);

}

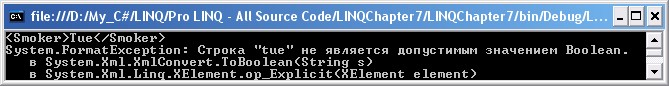
catch (Exception ex)

{

Console.WriteLine(ex);

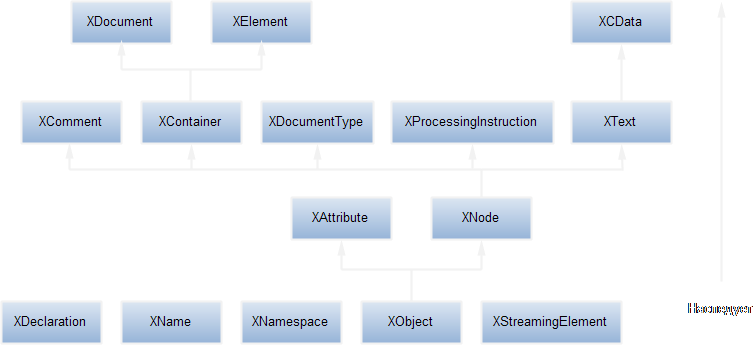
}

Обратите внимание, что в слове "True" специально допущена опечатка, чтобы инициировать исключение, которое покажет, где именно происходило приведение. Повезет ли? Нажмем <Ctrl+F5>, чтобы проверить:



Как видите, исключение было сгенерировано в методе System.Xml.XmlConvert.ToBoolean.

С новым API-интерфейсом LINQ to XML пришла новая объектная модель, содержащая множество новых классов, находящихся в пространстве имен System.Xml.Linq. Один из них — статический класс, в котором находятся расширяющие методы LINQ to XML; два класса компараторов — **XNodeDocumentOrderComparer** и **XNodeEqualityComparer**; остальные классы используются для построения деревьев XML. Эти остальные классы показаны на диаграмме, представленной на рисунке:



Следует отметить некоторые интересные вещи:

* Три из этих классов, XObject, XContainer и XNode, являются абстрактными, так что конструировать их объекты никогда не придется.
* Атрибут XAttribute не наследуется от узла XNode. Фактически, это вообще не узел, а совершенно другого типа класс, который на самом деле представляет пару "ключ-значение".
* Потоковые элементы XStreamingElement не имеют отношения наследования с элементом XElement.
* Классы XDocument и XElement — единственные классы, имеющие узлы, унаследованные от XNode.

Все эти классы используются для построения деревьев XML. Самое примечательное состоит в том, что применяется класс XElement, поскольку, как уже упоминалось, в LINQ to XML центральную роль играет элемент, в противоположность документу в W3C XML DOM.