

# Урок №8

## Содержание

<a href="#">XML-парсеры SAX и DOM.....</a>	<a href="#">2</a>
<a href="#">Создание и чтение XML документов.....</a>	<a href="#">5</a>
<a href="#">Класс XmlDocument.....</a>	<a href="#">5</a>
<a href="#">Класс XmlTextReader.....</a>	<a href="#">11</a>
<a href="#">Класс XmlValidatingReader.....</a>	<a href="#">15</a>
<a href="#">Класс XmlTextWriter.....</a>	<a href="#">19</a>
<a href="#">Использование Xpath.....</a>	<a href="#">20</a>
<a href="#">Оси.....</a>	<a href="#">24</a>
<a href="#">Системные функции.....</a>	<a href="#">25</a>
<a href="#">Функции с множествами.....</a>	<a href="#">26</a>
<a href="#">Строковые функции.....</a>	<a href="#">27</a>
<a href="#">Логические функции.....</a>	<a href="#">28</a>
<a href="#">Числовые функции.....</a>	<a href="#">29</a>
<a href="#">Использование XSL (XSLT).....</a>	<a href="#">37</a>
<a href="#">Преобразование XML в HTML на клиенте.....</a>	<a href="#">39</a>
<a href="#">Преобразование XML в HTML на сервере.....</a>	<a href="#">42</a>
<a href="#">Домашнее задание.....</a>	<a href="#">45</a>

---

## XML-парсеры SAX и DOM

XML-парсеры или XML-анализаторы — это программы, способные прочитывать XML-документы и извлечь из них данные, не заставляя вас разбирать синтаксис XML вручную. Большинство XML-анализаторов реализует один из двух популярных API: **DOM** или **SAX**. **DOM** является сокращением от **Document Object Model** (объектная модель документа) и описан по адресу <http://www.w3.org/TR/DOM-Level-2-Core/>. **SAX** — это **Simple API for XML** (простой API для XML) и является неофициальным (не утвержденным W3C) стандартом, сформировавшимся в результате усилий сообщества разработчиков на **Java** (см. его описание на <http://www.saxproject.org>). Оба API определяют программный интерфейс, позволяющий абстрагироваться от физической природы XML-документов, но используют для этого разные подходы.

**SAX** — это API на основе событий. Вы передаете SAX-анализатору один или несколько интерфейсов, содержащих predetermined набор методов обратного вызова, и по мере просмотра документа анализатор вызывает их, давая вам знать, что он нашел. Рассмотрим XML-документ:

```
<Greeting>Hello, world</Greeting>
```

Приложение, которое читает этот документ с помощью SAX-анализатора, реализует predetermined интерфейс, содержащий, помимо прочих, методы **startDocument**, **endDocument**, **startElement**, **endElement** и **characters**. По мере просмотра документа анализатор вызывает их в таком порядке:

**startDocument** // Сигнализирует о начале документа.

**startElement** // Сигнализирует о начале элемента Greeting,

**characters** // Передает "Hello, world".

...

**endElement** // Сигнализирует о конце элемента Greeting.

**endDocument** // Сигнализирует о конце документа.

При вызовах **startElement** и **endElement** передаются имена элементов. Многоточие после **characters** указывает, что **characters** вызывается неопределенное число раз. Некоторые анализаторы могут вызвать его один раз и передать «Hello, world» одним куском, другие же могут вызывать его несколько раз и передать «Hello, world» маленькими порциями. **SAX** очень хорош при разборе больших документов, так как не требует считывания в память всего документа сразу. Основной недостаток

**SAX** в том, что это API просмотра входного потока только в прямом направлении: вы не можете произвольно перемещаться по документу. Кроме того, нельзя легко определить отношения между его элементами, так как обратный вызов **SAX**-анализатора предоставляет крайне мало информации о контексте, в котором этот вызов происходит.

**DOM** является альтернативным API, который считывает документ в память и поддерживает произвольный доступ к его частям. **DOM** представляет собой дерево, отображающее структуру XML-документа. Элемент **documentElement** является верхним уровнем этого дерева. Этот элемент имеет один или несколько дочерних элементов **childNodes**, представляющих ветви дерева. Модель интерфейса на основе узлов используется для получения доступа к отдельным элементам дерева узлов.

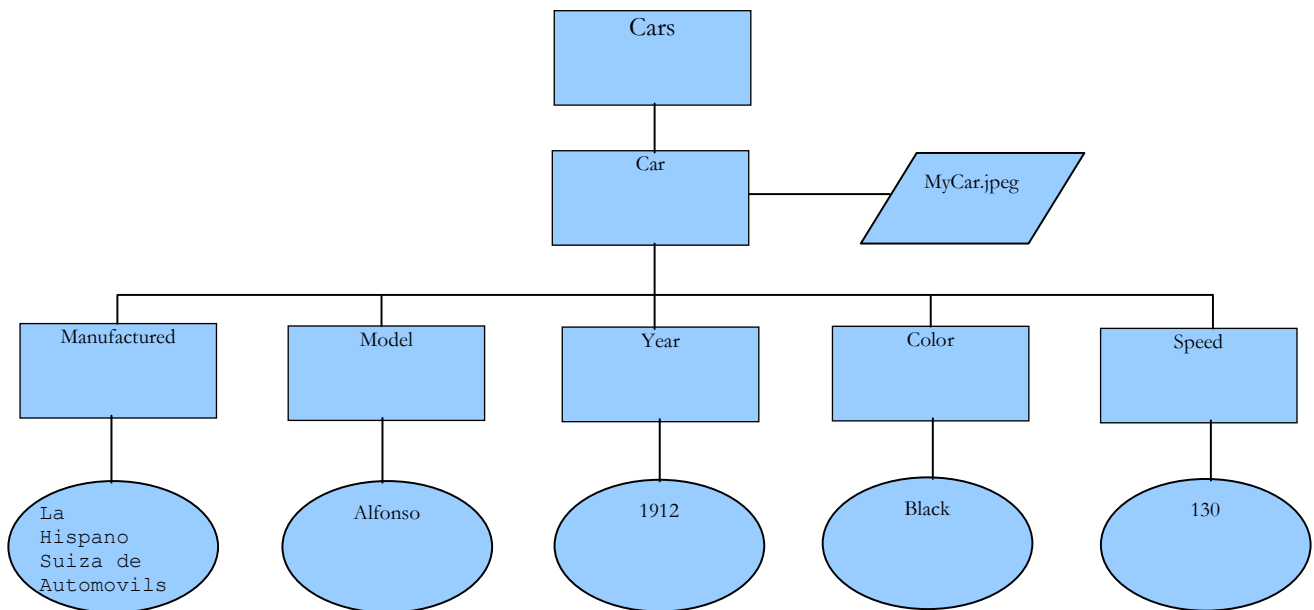
**Microsoft** предоставляет бесплатный **DOM**-анализатор в DLL — **MSXML.dll**, известный более как анализатор **MSXML** или просто **MSXML**. (Последние версии **MSXML** поддерживают также и **SAX**.) **MSXML** представляет XML-документы в виде объектной модели, соответствующей **DOM Level 2**. Отдельные части документа — элементы, атрибуты, текст и т. д. — представлены в виде узлов. На рисунке ниже показано дерево узлов, создаваемое **MSXML** в памяти для следующего XML-документа (файл Cars.xml):

```
<?xml version="1.0"?>
<Cars>
  <Car Image="MyCar.jpeg">
    <Manufactured>La Hispano Suiza de Automovils</Manufactured>
    <Model>Alfonso</Model>
```

```

<Year>1912</Year>
<Color>Black</Color>
<Speed>130</Speed>
</Car>
</Cars>

```



### DOM-представление простого XML-документа

Каждый блок диаграммы представляет узел. Прямоугольники соответствуют узлам элементов (элементов XML-документа), эллипсы — текстовым узлам (текстовые данные внутри элементов), а параллелограммы — атрибутам. Если бы документ содержал команды обработки или другие конструкции XML, они также были бы представлены узлами дерева. Каждый узел — это объект, предоставляющий методы и свойства для обхода дерева и извлечения содержимого. Так, каждый узел имеет свойство **hasChildNodes**, позволяющее определить, есть ли у него дочерние узлы, и свойства **firstChild** и **lastChild**, которые возвращают ссылки на дочерние узлы.

В библиотеке классов .NET Framework (FCL) есть маленький удобный класс **XmlDocument**, который предоставляет управляемую версию реализации **DOM** и делает анализ XML- документов поистине простым. При-

мер программы, написанной на C#, использующей этот класс показан ниже (проект ReadXml):

```
using System;
using System.Xml;
class MyApp
{
    static void Main()
    {
        XmlDocument doc = new XmlDocument();
        doc.Load("../..//..//..//Cars.xml");
        XmlNodeList nodes = doc.GetElementsByTagName("Car");
        foreach (XmlNode node in nodes)
        {
            Console.WriteLine("{0} {1}", node["Manufactured"].InnerText,
                               node["Model"].InnerText);
        }
    }
}
```

Метод **GetElementsByTagName** извлекает коллекцию элементов с заданным HTML-тегом.

## **Создание и чтение XML документов**

Пространство имен FCL **System.Xml** предоставляет разнообразные классы для чтения и генерации XML-документов. Если Вам удобнее работать с DOM, есть класс **XmlDocument** — у него те же возможности, что и у **MSXML**, но он гораздо проще в использовании. Если вы предпочитаете читать XML-документы как поток данных, то можете задействовать **XmlTextReader** или **XmlValidatingReader**, который поддерживает схемы. Дополняющий класс **XmlTextWriter** упрощает процесс создания XML- документов. Рассмотрим эти классы подробнее.

### **Класс XmlDocument**

**XmlDocument** реализует программный интерфейс к XML-документам, соответствующий спецификации **DOM Level 2 Core**. Он представляет

документ в виде перевернутого дерева узлов с корневым элементом или элементом-документом наверху.

Каждый узел является экземпляром класса ***XmlNode***, реализующим методы и свойства для обхода **DOM**-деревьев, считывания и изменения содержимого узлов, добавления и удаления узлов и др. ***XmlDocument*** является производным от ***XmlNode*** и добавляет собственные методы и свойства, которые поддерживают загрузку и сохранение документов, создание новых узлов и другие операции.

Следующие операторы создают объект ***XmlDocument*** и инициализируют его содержимым Cars.xml:

```
XmlDocument doc = new XmlDocument ();  
doc.Load ("Cars.xml");
```

***Load*** разбирает указанный XML-документ и строит его представление в памяти. Если документ не является правильно оформленным, генерируется ***XmlException***.

За успешным вызовом ***Load*** часто следует чтение значения свойства ***DocumentElement*** объекта ***XmlDocument***. ***DocumentElement*** возвращает ссылку на ***XmlNode*** для корневого элемента, который является начальной точкой обхода **DOM**-дерева.

Свойство ***HasChildNodes*** позволяет определить, есть ли у данного узла (в том числе у корневого) потомки. Для доступа к потомкам узла служит свойство ***ChildNodes*** — оно возвращает набор узлов ***XmlNodeList***. ***HasChildNodes*** и ***ChildNodes*** позволяют выполнять просмотр всех узлов дерева рекурсивно. Следующий фрагмент загружает XML-документ и выводит список узлов в консольное окно (проект XmlDocument):

```
using System;  
using System.Xml;  
class MyApp  
{  
    static void Main()  
    {  
        XmlDocument doc = new XmlDocument();  
        doc.Load("../..//..//..//Cars.xml");  
        OutputNode (doc.DocumentElement);  
    }  
}
```

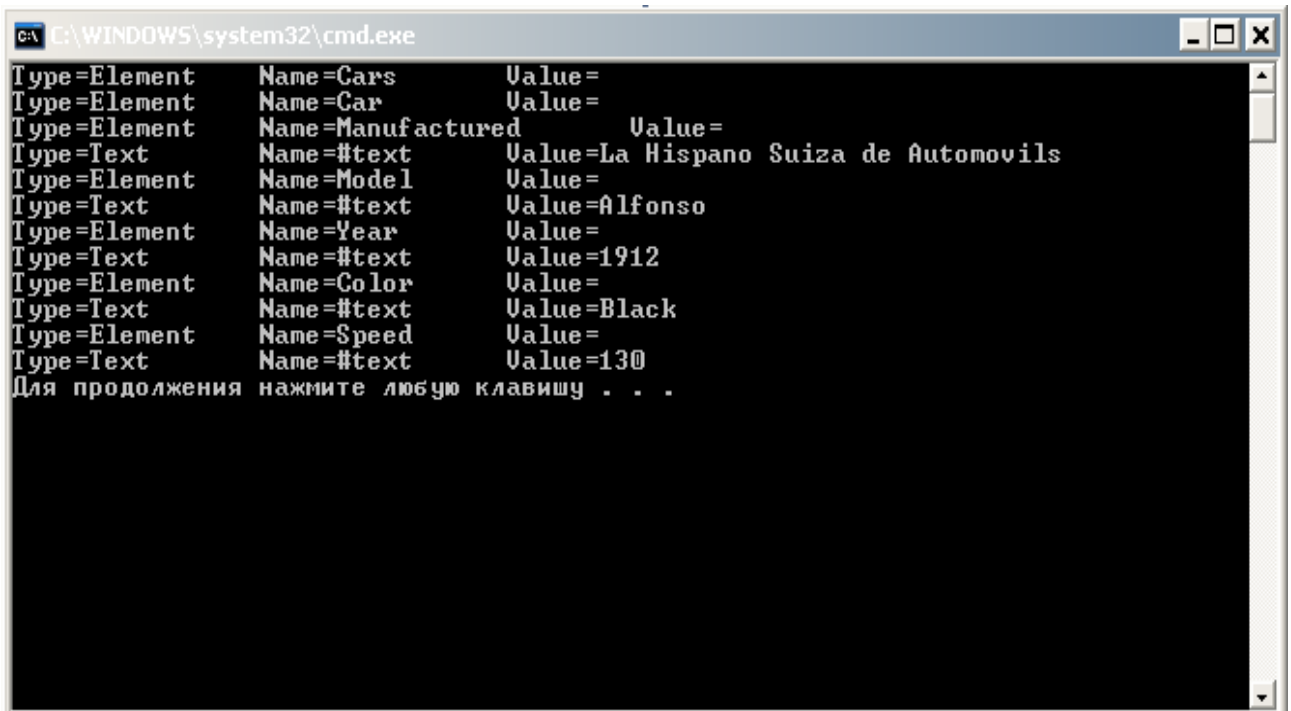
```

    }

    static void OutputNode(XmlNode node)
    {
        Console.WriteLine ("Type={0}\tName={1}\tValue={2}", node.NodeType,
node.Name, node.Value);
        if (node.HasChildNodes)
        {
            XmlNodeList children = node.ChildNodes;
            foreach (XmlNode child in children)
                OutputNode (child);
        }
    }
}

```

Для xml файла Cars.xml генерируются такие результаты:



```

C:\WINDOWS\system32\cmd.exe
Type=Element      Name=Cars          Value=
Type=Element      Name=Car            Value=
Type=Element      Name=Manufactured   Value=
Type=Text         Name=#text          Value=La Hispano Suiza de Automovils
Type=Element      Name=Model          Value=
Type=Text         Name=#text          Value=Alfonso
Type=Element      Name=Year            Value=
Type=Text         Name=#text          Value=1912
Type=Element      Name=Color           Value=
Type=Text         Name=#text          Value=Black
Type=Element      Name=Speed           Value=
Type=Text         Name=#text          Value=130
Для продолжения нажмите любую клавишу . . .

```

В первой колонке узлы элементов соответствуют элементам XML-документа, а узлы текста — тексту, связанному с этими элементами. Ниже перечислены все возможные типы узлов, представленные членами перечислимого типа ***XmlNodeType***.

Узлы **Whitespace** представляют «несущественные» пустые промежутки, т.е. пустые промежутки, расположенные между элементами разметки и, таким образом, ничего не добавляющие к содержимому документа и не появляющиеся среди узлов документа, если только перед вызовом **Load** не установить в **true** свойство **PreserveWhitespace** объекта **XmlDocument**, по умолчанию равное **false**.

XmlNodeType	Пример
Attribute	Attribute <Car Image="MyCar.jpeg">
CDATA	<![CDATA["This is character data"]>
Comment	<!-- This is a comment -->
Document	<Cars>
DocumentType	<!DOCTYPE Cars SYSTEM "Cars.dtd">
Element	<Car>
Entity	<!ENTITY filename "Strats.xml">
EntityReference	&lt;
Notation	<!NOTATION GIF89a SYSTEM "gif">
ProcessingInstruction	<?xml-stylesheet type="text/xsl"
Text	href="Cars.xsl"?>
Whitespace	<Model>Alfonso</Model>
XmlDeclaration	<Manufactured/>\r\n<Model/> <?xml version=" 1.0"?>

Заметьте: приведенные выше результаты работы программы не содержат узлов атрибутов, хотя в документе есть два элемента с атрибутами. Дело в том, что атрибуты обрабатываются особым образом. Они присутствуют не в списке, возвращаемом свойством **ChildNodes**, а в списке, который возвращается свойством **Attributes**. Вот как изменить метод **OutputNode**, чтобы атрибуты выводились наравне с другими типами узлов:

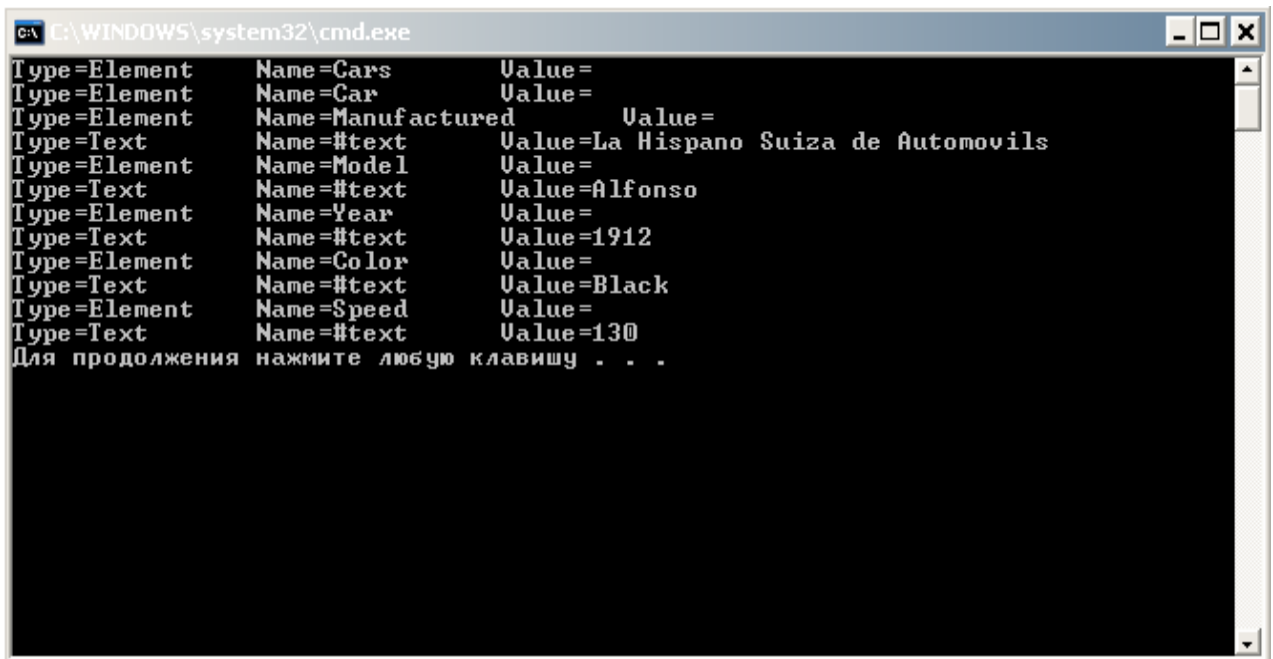


```

static void OutputNode (XmlNode node)
{
    Console.WriteLine ("Type={0}\tName={1}\tValue={2}",
        node.NodeType, node.Name, node.Value);
    if (node.Attributes != null)
    {
        foreach (XmlAttribute attr in node.Attributes)
            Console.WriteLine("Type= {0}\tName={1}\tValue={2}",
                attr.NodeType, attr.Name, attr.Value);
    }
    if (node.HasChildNodes)
    {
        foreach (XmlNode child in node.ChildNodes)
            OutputNode (child);
    }
}

```

Для xml файла Cars.xml новая версия функции OutputNode генерирует такие результаты:



```

C:\WINDOWS\system32\cmd.exe
Type=Element      Name=Cars          Value=
Type=Element      Name=Car            Value=
Type=Element      Name=Manufactured   Value=
Type=Text         Name=#text         Value=La Hispano Suiza de Automovils
Type=Element      Name=Model          Value=
Type=Text         Name=#text         Value=Alfonso
Type=Element      Name=Year           Value=
Type=Text         Name=#text         Value=1912
Type=Element      Name=Color          Value=
Type=Text         Name=#text         Value=Black
Type=Element      Name=Speed          Value=
Type=Text         Name=#text         Value=130
Для продолжения нажмите любую клавишу . . .

```

Свойства **NodeType**, **Name** и **Value** объекта **XmlNode** обеспечивают доступ к типу, имени и значению соответствующего узла. Для узлов некоторых типов (например, элементов) **Name** имеет смысл, а **Value** — нет. В других случаях (в частности, для текстовых узлов) имеет смысл

**Value**, а не **Name**. Бывает также (пример тому атрибуты), что имеет смысл и **Name**, и **Value**. **Name** возвращает квалифицированное имя узла, в которое входит и префикс, если он задан (например, *win:Car*). Свойство **LocalName** возвращает имена без префиксов.

Чтобы найти конкретный узел или группу узлов нет нужды просматривать все узлы документа. Методы **GetElementsByTagName**, **SelectNodes** и **SelectSingleNode** класса **XmlDocument** позволяют отобрать нужные узлы. В примере программы ReadXml.cs

**GetElementsByTagName** служит для того, чтобы быстро получить **XmlNodeList**, содержащий все узлы *Car* в документе. **SelectNodes** и **SelectSingleNode** исполняют выражения **XPath**. **XPath** мы рассмотрим ниже. **XmlDocument** позволяет не только читать, но и изменять XML-документы. Следующий код открывает *Cars.xml*, удаляет первый элемент *Car*, добавляет новый элемент *Motorcycle* и сохраняет результат в файле *Motorcycle.xml* (проект XmlDocument):

```
// Удалить первый элемент Cars
XmlNode root = doc.DocumentElement;
root.RemoveChild(root.FirstChild);
// Создать узлы элементов.
XmlNode bike = doc.CreateElement("Motorcycle");
XmlNode elem1 = doc.CreateElement("Manufactured");
XmlNode elem2 = doc.CreateElement("Model");
XmlNode elem3 = doc.CreateElement("Year");
XmlNode elem4 = doc.CreateElement("Color");
XmlNode elem5 = doc.CreateElement("Engine");
// Создать текстовые узлы
XmlNode text1 = doc.CreateTextNode("Harley-Davidson Motor Co. Inc.");
XmlNode text2 = doc.CreateTextNode("Harley 20J");
XmlNode text3 = doc.CreateTextNode("1920");
XmlNode text4 = doc.CreateTextNode("Olive");
XmlNode text5 = doc.CreateTextNode("37 HP");
// Присоединить текстовые узлы к узлам элементов
elem1.AppendChild(text1);
elem2.AppendChild(text2);
elem3.AppendChild(text3);
```

```
elem4.AppendChild(text4);
elem5.AppendChild(text5);
// Присоединить узлы элементов к узлу bike
bike.AppendChild(elem1);
bike.AppendChild(elem2);
bike.AppendChild(elem3);
bike.AppendChild(elem4);
bike.AppendChild(elem5);
// Присоединить узел bike к корневому узлу
root.AppendChild(bike);
// Сохранить измененный документ
doc.Save("Motorcycle.xml");
// Сохранить измененный документ
doc.Save("Motorcycle.xml");
```

## Класс XmlTextReader

Если нужно просто считать XML и его структура интересует вас меньше, чем содержимое, то можно использовать FCL-класс ***XmlTextReader***. Класс ***XmlTextReader***, как и ***XmlDocument***, относится к пространству имен ***System.Xml***, предоставляет быстрый способ последовательного просмотра XML-документа. Как и ***SAX***, он основан на понятии потока данных с возможностью чтения только вперед. Этот класс эффективнее ***XmlDocument*** по затратам памяти, особенно для больших документов, так как считывает весь документ сразу. Кроме того, он позволяет еще проще, чем ***XmlDocument***, сканировать документ в поиске элементов, атрибутов и т.п. Использовать ***XmlTextReader*** очень просто. Сначала из файла, URL или другого источника данных создается объект ***XmlTextReader***, затем последовательно вызывается ***XmlTextReader.Read***, пока не будет найдено нужное содержимое или не будет достигнут конец документа. Каждый вызов ***Read*** продвигает воображаемый курсор на следующий узел документа. Свойства ***XmlTextReader***, такие как ***NodeType.Name***, ***Value*** и ***AttributeCount***, возвращают информацию о текущем узле. Методы ***GetAttribute***,

***MoveToFirstAttribute*** и ***MoveToNextAttribute*** позволяют обращаться к атрибутам текущего узла, если таковые имеются.

Следующий код создает ***XmlTextReader*** для файла Cars.xml и просматривает узел за узлом весь файл (проект XmlTextReader):

```
using System;
using System.Xml;
class MyApp
{
    static void Main()
    {
        XmlTextReader reader = null;
        try
        {
            reader = new XmlTextReader("../..//..//..//Cars.xml");
            reader.WhitespaceHandling = WhitespaceHandling.None;
            while (reader.Read())
            {
                Console.WriteLine("Type={0}\t\tName={1}\t\tValue={2}",
                    reader.NodeType, reader.Name, reader.Value);
            }
        }
        finally
        {
            if (reader != null)
                reader.Close();
        }
    }
}
```

Для XML-документа Cars.xml эта программа выводит такие результаты:

```

C:\WINDOWS\system32\cmd.exe
Type=XmlDeclaration      Name=xml      Value=version="1.0"
Type=Element             Name=Cars     Value=
Type=Element             Name=Car      Value=
Type=Element             Name=Manufactured Value=
Type=Text                Name=        Value=La Hispano Suiza de Automovils
Type=EndElement          Name=Manufactured Value=
Type=Element             Name=Model    Value=
Type=Text                Name=        Value=Alfonso
Type=EndElement          Name=Model    Value=
Type=Element             Name=Year     Value=
Type=Text                Name=        Value=1912
Type=EndElement          Name=Year     Value=
Type=Element             Name=Color    Value=
Type=Text                Name=        Value=Black
Type=EndElement          Name=Color    Value=
Type=Element             Name=Speed    Value=
Type=Text                Name=        Value=130
Type=EndElement          Name=Speed    Value=
Type=EndElement          Name=Car      Value=
Type=EndElement          Name=Cars     Value=
Для продолжения нажмите любую клавишу . . .

```

Обратите внимание на узлы **EndElement**. **XmlTextReader** в отличие от **XmlDocument** считает начальные и завершающие тэги элемента отдельными элементами. **XmlTextReader** также возвращает узлы пустых промежутков, если не указано иное, Установка его свойства **WhitespaceHandling** в **WhitespaceHandlingNone** подавляет выдачу пустых промежутков. **XmlTextReader**, как и **XmlDocument** не возвращает атрибуты как часть нормального процесса просмотра, просматривать узлы атрибутов нужно отдельно. Вот измененный вариант нашего примера, который выводит узлы атрибутов наравне с другими узлами (проект XmlTextReader):

```

using System;
using System.Xml;
class MyApp
{
    static void Main()
    {
        XmlTextReader reader = null;
        try
        {
            reader = new XmlTextReader("../..//..//..//Cars.xml");

```

```
reader.WhitespaceHandling = WhitespaceHandling.None;
while (reader.Read())
{
    Console.WriteLine("Type={0}\t\tName={1}\t\tValue={2}",
        reader.NodeType, reader.Name, reader.Value);
    if (reader.AttributeCount > 0)
    {
        while (reader.MoveToNextAttribute())
        {
            Console.WriteLine("Type={0}\t\tName={1}\t\tValue={2}",
                reader.NodeType, reader.Name, reader.Value);
        }
    }
}
finally
{
    if (reader != null)
        reader.Close();
}
}
```

***XmlTextReader*** часто применяют для извлечения из XML значений заданных узлов. Следующий код отыскивает все элементы Car с атрибутом *Image* и выводит на консоль значения этих атрибутов:

```
using System;
using System.Xml;
class MyApp
{
    static void Main()
    {
        XmlTextReader reader = null;
        try
        {
            reader = new XmlTextReader("../..//..//..//Cars.xml");
            reader.WhitespaceHandling = WhitespaceHandling.None;
            while (reader.Read())
            {
                if (reader.NodeType == XmlNodeType.Element &&
```

```
        reader.Name == "Car" && reader.AttributeCount > 0)
    {
        while (reader.MoveToNextAttribute())
        {
            if (reader.Name == "Image")
            {
                Console.WriteLine(reader.Value);
                break;
            }
        }
    }
}
finally
{
    if (reader != null)
        reader.Close();
}
}
```

Для Cars.xml результат будет таким:

MyCar.jpeg

При завершении работы с ***XmlTextReader*** важно его закрыть, чтобы он в свою очередь мог закрыть источник данных. Вот почему все примеры этого раздела вызывают ***Close*** для ***XmlTextReader*** и делают это в блоках, ***finally***.

## Класс XmlValidatingReader

Класс ***XmlValidatingReader*** является производным от ***XmlTextReader***. Он имеет одно важное свойство, отсутствующее у ***XmlTextReader***, — возможность проверки допустимости XML-документов. Класс поддерживает три типа схем: **DTD**, **XSD** и **XML-Data Reduced (XDR)**. Его свойство ***Schemas*** содержит схему (или схемы), на соответствие которым проверяется документ, а свойство ***ValidationType*** задает тип схемы. По умолчанию ***ValidationType*** равно ***ValidationTypeAuto***, что позволяет

**XmlValidatingReader** определять тип схемы по переданному ему документу схемы. Установка **ValidationType** в **ValidationTypeNone** дает анализатор без проверки — эквивалент **XmlTextReader**.

**XmtValidatingReader** не принимает как входной параметр имя файла или URL, однако вы можете передать имя файла или URL **XmlTextReader** и уже для него создать **XmlValidatingReader**. Следующие операторы создают **XmlValidatingReader** и передают ему XML-документ и документ схемы:

```
XmlTextReader nvr = new XmlTextReader ("Cars.xml");  
XmlValidatingReader reader = new XmlValidatingReader (nvr);  
reader.Schemas.Add ("", "Cars.xsd");
```

Первый параметр метода **Add** задает целевое пространство имен, заданное документом схемы, если оно есть, Пустая строка означает, что схема не определяет целевое пространство имен.

Проверка документа столь же проста, как просмотр всех его узлов повторяющимися вызовами **XmlValidatingReader.Read**:

```
while (reader.Read ());
```

Если при этом обнаруживается ошибка правильности оформления, то генерируется **XmlException**. Если же обнаруживаются ошибки допустимости, то генерируются события **ValidationEventHandler**. Приложение, использующее **XmlValidatingReader**, может перехватывать эти события, зарегистрировав обработчик:

```
reader.ValidationEventHandler+=new ValidationEventHandler(OnValidationError);
```

Обработчик события получает параметр **ValidationEventArgs**, содержащий информацию о нарушении правил допустимости, включая его текстовое описание (**ValidationEventArgsMessage**) и **XmlSchemaException** (в **ValidationEventArgsException**). Последнее содержит дополнительную информацию об ошибке, такую как место документа, где она была обнаружена.

В коде ниже показан исходный текст консольного приложения Validate, выполняющего проверку соответствия XML-документов XML-схемам. Для



запуска введите имя программы, после которого укажите имя или URL XML-документа, а также имя или URL XML-схемы, например:

validate cars.xml cars.xsd

Для удобства пользователя Validate автоматически определяет целевое пространство имен схемы, необходимое для добавления схемы в набор **Schemas**, применяя **XmlTextReader** для разбора документа схемы. При этом используется тот факт, что **XSD** в отличие от **DTD** сами являются XML-документами, а значит, могут быть считаны XML-анализаторами (проект Validate).

```
using System;
using System.Xml;
using System.Xml.Schema;
class MyApp
{
    static void Main (string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Syntax; VALIDATE xmldoc schemadoc");
            return;
        }
        XmlValidatingReader reader = null;
        try
        {
            XmlTextReader nvr = new XmlTextReader (args[0]);
            nvr.WhitespaceHandling = WhitespaceHandling.None;
            reader = new XmlValidatingReader (nvr);
            reader.Schemas.Add (GetTargetNamespace (args[1]), args[1]);
            reader.ValidationEventHandler += new
                ValidationEventHandler (OnValidationError);
            while (reader.Read ());
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {

```

```
        if (reader != null)
            reader.Close();
    }
}

static void OnValidationError (object sender, ValidationEventArgs e)
{
    Console.WriteLine (e.Message);
}

public static string GetTargetNamespace (string src)
{
    XmlTextReader reader = null;
    try
    {
        reader = new XmlTextReader (src);
        reader.WhitespaceHandling = WhitespaceHandling.None;
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element &&
                reader.LocalName == "schema")
            {
                while (reader.MoveToNextAttribute ())
                {
                    if (reader.Name == "targetNamespace")
                        return reader.Value;
                }
            }
        }
        return "";
    }
    finally
    {
        if (reader != null)
            reader.Close ();
    }
}
```

## Класс XmlTextWriter

Класс **XmlDocument** можно применять для изменения, но не для создания XML- документов. Для этого служит класс **XmlTextWriter**. Его методы **Write** генерируют различные фрагменты XML, в том числе элементы, атрибуты, комментарии и др.

В следующем примере некоторые из этих методов служат для создания XML-документа Cars.xml, содержащего корневой элемент *Cars* и дочерний элемент *Car* (проект XmlTextWriter):

```
using System;
using System.Xml;
class MyApp
{
    static void Main()
    {
        XmlTextWriter writer = null;
        try
        {
            writer = new XmlTextWriter ("Cars.xml",
System.Text.Encoding.Unicode);
            writer.Formatting = Formatting.Indented;
            writer.WriteStartDocument ();
            writer.WriteStartElement ("Cars");
            writer.WriteStartElement ("Car");
            writer.WriteAttributeString ("Image", "MyCar.jpeg");
            writer.WriteElementString ("Manufactured", "La Hispano Suiza de
Automovils");
            writer.WriteElementString ("Model", "Alfonso");
            writer.WriteElementString ("Year", "1912");
            writer.WriteElementString ("Color", "Black");
            writer.WriteElementString ("Speed", "130");
            writer.WriteEndElement ();
            writer.WriteEndElement ();
        }

        finally
        {
            if (writer != null)
```

```

        writer.Close ();
    }
}

```

Вот что получается в результате:

```

<?xml version="1.0" encoding="utf-16" ?>
- <Cars>
- <Car Image="MyCar.jpeg">
    <Manufactured>La Hispano Suiza de Automovils</Manufactured>
    <Model>Alfonso</Model>
    <Year>1912</Year>
    <Color>Black</Color>
    <Speed>130</Speed>
  </Car>
</Cars>

```

Установка свойства **Formatting** в **Formatting.Indented** перед началом записи порождает отступы, которые вы видите в примере. Если этого не сделать, то не генерируются ни отступы, ни переводы строки. По умолчанию ширина отступов равна 2, а символ-заполнитель — пробел. Свойства **Indentation** и **IndentChar** позволяют изменить ширину отступов и символ-заполнитель.

## Использование Xpath

**XPath** (сокращение от **XML Path Language**) — это язык адресации фрагментов XML- документа. Слово «path» (путь) в его названии вызвано сходством между XML-путями и путями в файловой системе. Так, в файловой системе «\Book\Chap13» задает подкаталог Chap13 в каталоге Book — подкаталоге корневого каталога. В XML-документе «/Cars/Car» соответствует всем элементам *Car*, являющимся дочерними для корневого элемента *Cars*. «/Cars/Car» — это выражение **XPath**. Полностью выражения **XPath** описаны в спецификации Xpath (<http://www.w3.org/TR/xpath>).

**XPath** можно использовать различными способами. Ниже мы познакомимся с **XSLT (XSL Transformations)** — языком для преобразования

XML-документов из одного формата в другой. XSLT использует выражения **XPath** для задания узлов и их наборов. Другое распространенное применение **XPath** — извлечение данных из XML-документов. При этом **XPath** становится языком запросов — XML-эквивалентом SQL. W3C работает над официальным языком запросов **XQuery** (<http://www.w3.org/TR/xquery>), но на данный момент **XPath** - процессор — это лучший способ извлечения данных из XML-документов без прохода по **DOM**-деревьям вручную. XPath призван помочь обходить всевозможные деревья, получать необходимые элементы из другой ветви относительно точки обхода, распознавать предков, потомков, атрибуты элементов. Это полноценный язык навигации по дереву.

Для нахождения элемента(ов) в дереве документа используются пути адресации. Например, рассмотрим XML документ:

```
<html>

<body>

  <div>Первый слой

    <span>блок текста в первом слое</span>

  </div>

  <div>Второй слой</div>

  <div>Третий слой

    <span class="text">первый блок в третьем слое</span>

    <span class="text">второй блок в третьем слое</span>

    <span>третий блок в третьем слое</span>

  </div>

  <img />

</body>
```

---

---

```
</html>
```

Рассмотрим также путь адресации **/html/body/\*/span[@class]** (полный синтаксис имеет вид

**/child::html/child::body/child::\* /child::span[attribute::class]**) который вернет набор из двух элементов исходного документа (*<span class="text">первый блок в третьем слое</span>* и *<span class="text">второй блок в третьем слое</span>*). Путь делится на *шаги адресации* которые разделяются символом косая черта / . В свою очередь, каждый шаг адресации состоит из трех частей:

- ось (в данном примере `child::`), это обязательная часть;
- условие проверки узлов (в данном примере это имена элементов документа `html`, `body`, `span`, а символ `*` означает элемент с любым именем), это обязательная часть;
- предикат (в данном примере `attribute::class`), необязательная часть заключаемая в квадратные скобки в которой могут содержаться оси, условия проверки, функции, операторы (`+`, `-`, `<`, `>` и проч.).

Анализ ведется слева направо. Если первый символ это / , то путь адресации считается абсолютным. При этом за узел контекста на первом шаге берется корневой элемент (`html`). Контекст — это некая точка отсчета, относительно которой рассчитывается следующий шаг адресации. Поэтому на каждом шаге адресации мы получаем новый набор узлов документа, и этот набор становится контекстом для следующего шага адресации.

На втором шаге адресации (`child::body`) контекстом становится `html` элемент. Ось `child::` говорит о том, что необходимо найти все непосредственные потомки элемента `html`, а условие проверки `body` говорит о том, что в формируемый набор элементов нужно включить все узлы с именем `body`. В ходе второго шага адресации получаем набор узлов, состоящий всего из одного элемента `body`, который и становится элементом контекста для третьего шага.

Третий шаг адресации: `child::*` . Ось `child::` собирает все непосредственные потомки элемента `body`, а условие проверки `*` говорит о том, что в формируемый набор нужно включить элементы основного типа с любым именем. В ходе этого шага получаем набор узлов, состоящий из трех элементов `div` и одного элемента `img`.

Четвёртый шаг адресации: `child::span` . Теперь контекстом является набор из четырёх элементов. И следующий набор узлов создается в четыре прохода (за четыре итерации). При первой итерации узлом контекста становится первый `div`. Согласно заданной оси `child::` и правилу проверки `span`, в набор включаются непосредственные потомки `div`-а, имя которых равно `span`. При второй итерации в набор ничего добавлено не будет, так как у второго `div` нет потомков. Третья итерация добавит в набор сразу три элемента `span`, а четвёртая ничего не добавит, так как у элемента `img` нет потомков. Итак, в ходе проверки получен набор узлов, состоящий из четырёх элементов `span`. Это и будет контекстом для следующей обработки.

Следующего шага нет, поэтому будет производиться фильтрация отобранного набора. В этом и состоит отличие предикатов от шагов адресации. На каждом шаге адресации получаем новый набор, отталкиваясь от контекста, полученного на предыдущем шаге. В ходе же обработки предиката новый набор получается из текущего методом фильтрации, когда из набора исключаются узлы, не прошедшие условие проверки. В данном случае ось `attribute::` говорит о необходимости проверить, если ли у узлов контекста атрибуты, а условие `class` требует оставить лишь те узлы, у которых задан атрибут с именем `class`. Фильтрация происходит за четыре итерации, но в окончательный набор попадают только два элемента `span`.

## **Оси**

Оси это база языка XPath.

- **ancestor::**— Возвращает множество предков.
- **ancestor-or-self::**— Возвращает множество предков и текущий элемент.
- **attribute::**— Возвращает множество атрибутов текущего элемента.
- **child::**— Возвращает множество потомков на один уровень ниже.
- **descendant::**— Возвращает полное множество потомков.
- **descendant-or-self::**— Возвращает полное множество потомков и текущий элемент.
- **following::**— Возвращает необработанное множество, ниже текущего элемента.
- **following-sibling::**— Возвращает множество элементов на том же уровне, следующих за текущим.
- **namespace::**— Возвращает множество имеющее пространство имён (то есть присутствует атрибут xmlns).
- **parent::**— Возвращает предка на один уровень назад.
- **preceding::**— Возвращает множество обработанных элементов включая множество предков.
- **preceding-sibling::**— Возвращает множество элементов на том же уровне, предшествующих текущему.
- **self::**— Возвращает текущий элемент.

Существуют сокращения для некоторых осей, например:

- **attribute::**— можно заменить на «@»
- **child::**— часто просто опускают
- **descendant-or-self::**— можно заменить на «//»
- **parent::**— можно заменить на «..»
- **self::**— можно заменить на «.»

Дополнением к базе является набор функций, которые делятся на 5 групп:



## Системные функции

### **node-set document(object!, node-set?)**

Возвращает документ указанный в параметре object!.

### **string format-number(number, string, string?)**

Форматирует число согласно образцу указанному во втором параметре, третий параметр указывает именованный формат числа, который должен быть учтён.

### **string generate-id(node-set?)**

Возвращает строку, являющуюся уникальным идентификатором.

### **node-set key(string, object!)**

Возвращает множество с указанным ключом, аналогично функции id для идентификаторов.

### **string unparsed-entity-uri(string)**

Возвращает непроанализированный URI, если такового нет, возвращает пустую строку.

### **boolean element-available(string)**

Проверяет доступен ли элемент или множество указанное в параметре. Параметр рассматривается как XPath.

### **boolean function-available(string)**

Проверяет доступна ли функция указанная в параметре. Параметр рассматривается как XPath.

### **object! system-property(string)**

Параметры, возвращающие системные переменные, могут быть:

- \* **xsl:version**— возвращает версию XSLT процессора.
- \* **xsl:vendor**— возвращает производителя XSLT процессора.
- \* **xsl:vendor-url**— возвращает URL идентифицирующий производителя.

Если используется неизвестный параметр, функция возвращает пустую строку.

### **boolean lang(string)**

Возвращает истину если у текущего тега имеется атрибут `xml:lang`, либо родитель тега имеет атрибут `xml:lang` и в нем указан совпадающий строке символ.

### Функции с множествами

- `*` — обозначает *любое* имя или набор символов, `@*` — любой атрибут
- `$name` — обращение к переменной, где `name` — имя переменной или параметра.
- `[]` — дополнительные условия выборки
- `{ }` — если применяется внутри тега другого языка (например HTML), то XSLT процессор, то что написано в фигурных скобках рассматривает как XPath.
- `/` — определяет уровень дерева

#### **node-set node()**

Возвращает элемент(ы). Для этой функции часто используют заменитель `'*'`, но в отличие от звездочки — `node()` возвращает и *текстовые* элементы.

#### **node-set current()**

Возвращает множество из одного элемента, который является текущим. Если мы делаем обработку множества с условиями, то единственным способом дотянуться из этого условия до текущего элемента будет данная функция.

#### **number position()**

Возвращает позицию элемента в множестве. Корректно работает только в цикле `<xsl:for-each/>`

#### **number last()**

Возвращает номер последнего элемента в множестве. Корректно работает только в цикле `<xsl:for-each/>`

#### **number count(node-set)**

Возвращает количество элементов в `node-set`

**string name(node-set?)**

Возвращает полное имя первого тега в множестве.

**string namespace-uri(node-set?)**

Возвращает ссылку на url определяющий пространство имён.

**string local-name(node-set?)**

Возвращает имя первого тега в множестве, без пространства имён.

**node-set id(object!)**

Находит элемент с уникальным идентификатором

**Строковые функции**

**string text()**

Возвращает текстовое содержимое элемента. По сути возвращает объединенное множество *текстовых* элементов на *один* уровень ниже.

**string string(object?)**

Конвертирует объект в строку.

**string concat(string, string, string\*)**

Объединяет две или более строк

**number string-length(string?)**

Возвращает длину строки.

**boolean contains(string, string)**

Возвращает истину, если первая строка содержит вторую, иначе возвращает ложь.

**string substring(string, number, number?)**

Возвращает строку вырезанную из строки начиная с указанного номера, и если указан второй номер— количество символов.

**string substring-before(string, string)**

Если найдена вторая строка в первой, возвращает строку до первого вхождения второй строки.

**string substring-after(string, string)**

Если найдена вторая строка в первой, возвращает строку после первого вхождения второй строки.

**boolean starts-with(string, string)**

Возвращает истину если вторая строка входит в начало первой, иначе возвращает ложь.

**boolean ends-with(string, string)**

Возвращает истину если вторая строка входит в конец первой, иначе возвращает ложь.

**string normalize-space(string?)**

Убирает лишние и повторные пробелы, а так же управляющие символы, заменяя их пробелами.

**string translate(string, string, string)**

Заменяет символы первой строки, которые встречаются во второй строке, на соответствующие по позиции символам из второй строки символы из третьей строки. `translate(«bar», «abc», «ABC»)` вернет `BAr`.

### Логические функции

- **or** — логическое «или»
- **and** — логическое «и»
- **=** — логическое «равно»
- **< (&lt;)** — логическое «меньше»
- **> (&gt;)** — логическое «больше»
- **<= (&lt;=)** — логическое «меньше либо равно»
- **>= (&gt;=)** — логическое «больше либо равно»

**boolean boolean(object)**

Приводит объект к логическому типу

**boolean true()**

Возвращает истину.

**boolean false()**

Возвращает ложь.

**boolean not(boolean)**

Отрицание, возвращает истину если аргумент ложь и наоборот.

**Числовые функции**

- **+** — сложение
- **-** — вычитание
- **\*** — умножение
- **div** — деление
- **mod** — остаток от деления

**number number(object?)**

Переводит объект в число.

**number sum(node-set)**

Вернёт сумму множества, каждый тег множества будет преобразован в строку и из него получено число.

**number floor(number)**

Возвращает наибольшее целое число, не большее, чем аргумент.

**number ceiling(number)**

Возвращает наименьшее целое число, не меньшее, чем аргумент.

**number round(number)**

Округляет число по математическим правилам.

В FCL имеется **XPath** — процессор **System.Xml.XPath.XPathNavigator**. Но прежде чем мы его рассмотрим, приведем краткий обзор **XPath**. Строительными блоками **XPath** служат выражения. Самый распространенный тип выражения — *путь поиска* (**location path**). Следующий путь поиска задает все элементы *Car*, дочерние для корневого элемента *Cars*:

```
/Cars/Car
```

А этот путь соответствует всем атрибутам (не элементам) *Image*, принадлежащим элементам *Car*, дочерних для корневого элемента *Cars*:

```
/Cars/Car/@Image
```

Следующее выражение соответствует всем элемента *Car* в любом месте документа:

```
//Car
```

Префикс `//` очень полезен для поиска элементов документа независимо от их расположения.

**XPath** также поддерживает символы подстановки. Следующее выражение задает все элементы, дочерние для корневого элемента с именем

```
Cars:  
/Cars/*
```

А это возвращает все атрибуты элементов *Car*, причем последние могут располагаться в любом месте документа:

```
//Car/@*
```

Пути поиска могут быть абсолютными или относительными. Пути, начинающиеся с `/` или `//` — абсолютные, так как задают местоположение относительно корня. А те, что не начинаются с `/` или `//`, — относительные. Они указывают местоположение относительно текущего узла или *узла контекста* в документе **XPath**.

Компоненты пути поиска называются *шагами поиска* (location step). Этот путь поиска содержит два шага поиска:

```
/Cars/Car
```

Шаг поиска состоит из трех частей: оси, теста узла и нуля или нескольких предикатов. В общем виде шаг поиска выглядит так:

```
ось::тест-узла[предикат1][предикат2][...]
```

Ось описывает отношение между узлами. Среди других поддерживаются значения ***child*** (прямой потомок), ***descendant*** (потомок), ***descendant-or-self*** (потомок или сам), ***parent*** (родитель), ***ancestor*** (предок) и ***ancestor-or-self*** (предок или сам). Если ось не задана, то по умолчанию берется ***child***. Таким образом, выражение:

```
/Cars/Car
```

можно записать в виде:

```
/child::Cars/child::Car
```

Другие оси предоставляют иные способы задания путей поиска. Например, такое выражение соответствует всем элементам *Car* — потомкам корневого элемента:

```
/descendant::Car
```

А это выражение задает все элементы *Car*, которые являются потомками корневого элемента или корневыми элементами:

```
/descendant-or-self:: Car
```

На самом деле `//` — это сокращение для `/descendant-or-self`. Таким образом, выражение:

```
//Car
```

эквивалентно приведенному выше. Аналогично `@` — сокращение для *attribute*.

Оператор:

```
//Car/@*
```

можно записать и так:

```
//Car/attribute::*
```

Большинство разработчиков предпочитает сокращенный вариант, однако процессоры, совместимые с **XPath 1.0**, поддерживают оба варианта.

Если предикат присутствует в пути поиска, то это часть пути, заключенная в квадратные скобки. Предикаты — это просто фильтры. Например, следующее выражение задает все элементы *Car* документа:

```
//Car
```

Тогда как следующее выражение с помощью предиката отбирает лишь те элементы *Car*, что имеют атрибут *Image*:

```
//Car[@Image]
```

А это выражение соответствует всем элементам *Car*, атрибуту *Image* которых присвоено значение «MyCar.jpeg»:

```
//Car[@Image = "MyCar.jpeg"]
```

Предикаты могут использовать операторы сравнения: `<`, `>`, `=`, `!=`, `<=` и `>=`. Следующее выражение задает те элементы *Car*, в которых элементы *Year* задают год после 1980:

```
//Car[Year > 1980]
```

В предикатах также допускаются операторы *and* и *or*. Следующее выражение отбирает автомобили, произведенные Honda после 1990г.:

```
//Car[Year > 1990][Manufactured = "Honda"]
```

А это объединяет эти два предиката в один с использованием оператора *and*:

```
//Car[Year > 1990 and Manufactured = "Honda"]
```

Если заменить *and* на *or*, то получим автомобили, произведенные или Honda, или после 1990 г.:

```
//Car[Year > 1990 or Make = "Honda"]
```

**XPath** также поддерживает набор встроенных функций, которые часто (но не всегда) применяются в предикатах. Следующее выражение задает все элементы *Car*, имеющие элементы *Manufactured*, текст которых начинается с буквы H. Для этого предикат вызывает функцию **starts-with**:

```
//Car[starts-with (Manufactured, "H")]
```

Это выражение использует функцию **text** для отбора всех текстовых узлов, связанных с элементами *Manufactured*, являющимися дочерними элементами для элементов *Car*. Как и **DOM**, **XPath** рассматривает связанный с элементом текст как отдельный узел:

```
//Car/Manufactured/text
```

Функции **starts-with** и **text** — лишь две из многих функций, поддерживаемых **XPath**. Полный список функций см. в спецификации **XPath**.

В результате исполнения пути поиска **XPath**-процессором возвращается набор узлов (**node set**). Для представления XML-содержимого **XPath**, как и **DOM**, применяет древовидную организацию наборов узлов. Допустим, для нашего XML-документа выполнен такой путь поиска:

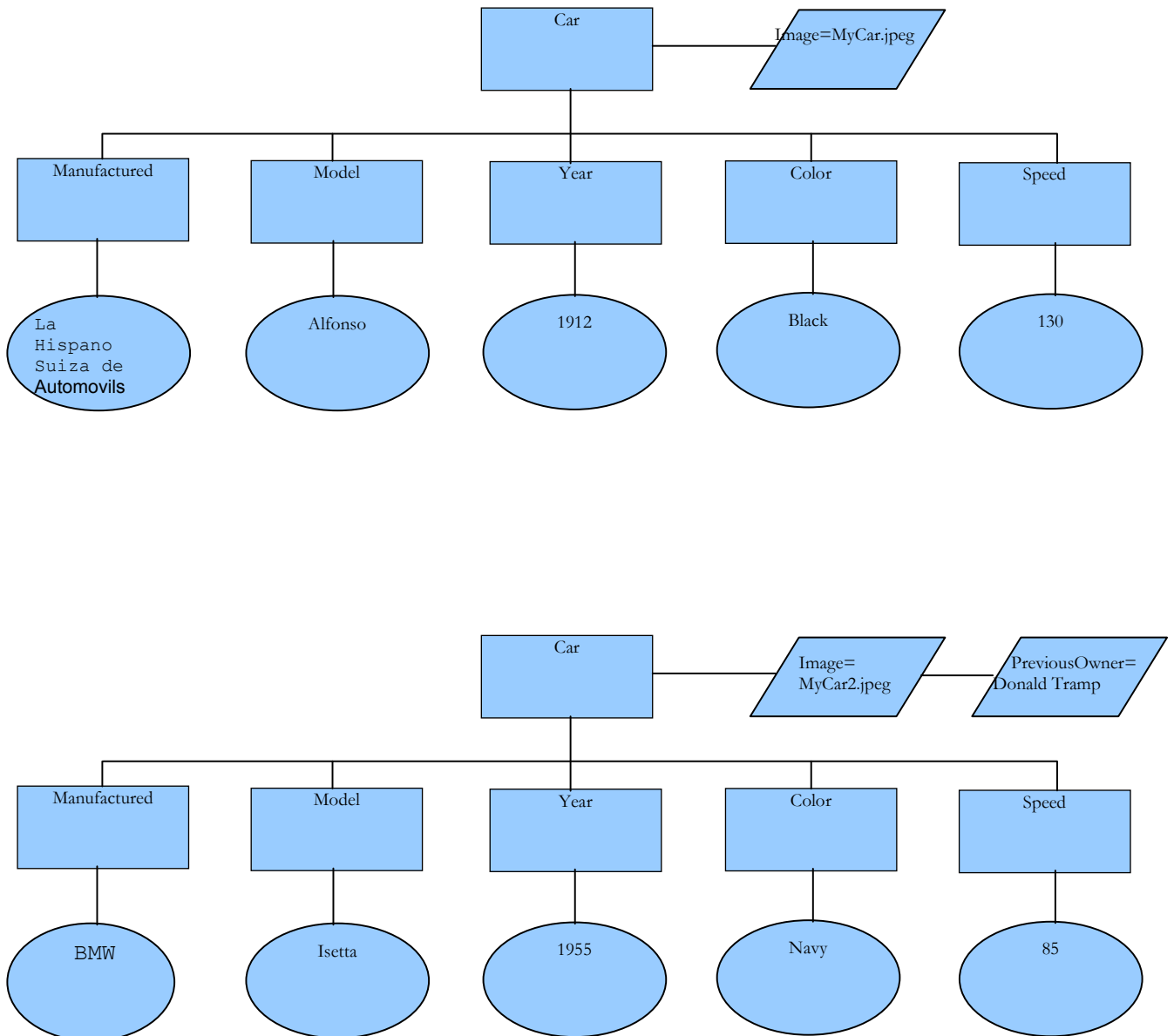
```
//Car
```

Полученный набор узлов содержит два узла, каждый из которых представляет элемент *Car*. Каждый элемент *Car* — это корень дерева узлов, содержащих узлы дочерних элементов *Manufactured*, *Model*, *Year*, *Color* и *Speed* (как показано на рисунке ниже).

Каждый дочерний узел является родительским для текстового узла, содержащего текст элемента. Типы узлов XPath определены независимо от



типов узлов **DOM**, хотя и имеют с ними много общего. В **XPath** меньше типов узлов, чем в **DOM**, что делает типы узлов **XPath** функциональным подмножеством типов узлов DOM.



Набор узлов, полученный по выражению XPath.

Пространство имен **System.Xml.XPath** библиотеки классов **.NET Framework** содержит классы, позволяющие использовать **XPath** в управляемых приложениях. Главными среди этих классов являются *XPathDocument* (представляет XML-документы, к которым будут выполняться **XPath**-запросы), *XPathNavigator* (представляет механизм для выполнения **XPath**-запросов) и *XPathNodeIterator* (представляет

наборы узлов, сгенерированные **XPath**-запросами и позволяет выполнять перебор их содержимого).

Первым шагом при выполнении запросов **XPath** для XML-документов является создание **XPathDocument** для самого XML-документа.

**XPathDocument** предоставляет различные конструкторы, которые позволяют инициализировать его из потока данных, по URL, из файла, с помощью **TextReader** или **XmlReader**. Этот оператор создает объект **XPathDocument** и инициализирует его содержимым Cars.xml:

```
XPathDocument doc = new XPathDocument ("../../../../../Cars.xml");
```

Второй шаг состоит в создании **XPathNavigator** для **XPathDocument**.

Для этого **XPathDocument** предоставляет метод **CreateNavigator**. Следующий оператор создает **XPathNavigator** для **XPathDocument**, созданного выше:

```
XPathNavigator nav = doc.CreateNavigator();
```

Последний шаг — собственно исполнение запроса. **XPathNavigator** предоставляет пять методов для исполнения **XPath**-запросов. Самыми важными из них - **Evaluate** и **Select**. **Evaluate** вычисляет любое выражение **XPath**. Он возвращает **Object** общего вида, который может быть строкой, вещественным числом, булевым значением или объектом

**XPathNodeIterator** в зависимости от выражения и типа возвращаемых им данных, **Select** работает только с выражениями, возвращающими наборы узлов, а значит, представляет собой удобное средство вычисления путей поиска. Он всегда возвращает **XPathNodeIterator**, представляющий набор узлов **XPath**.

Следующий оператор с помощью **Select** создает набор узлов, который соответствует всем узлам, удовлетворяющим выражению «**//Car**»:

```
XPathNodeIterator iterator = nav.Select ("//Car");
```

**XPathNodeIterator** — это простой класс, позволяющий выполнить перебор узлов, возвращенных в составе набора. Количество узлов в наборе возвращается свойством **Count**:

```
Console.WriteLine ("Select returned {0} nodes", iterator.Count);
```

Метод **MoveNext** класса **XPathNodeIterator** позволяет просмотреть набор узлов по одному за раз. При просмотре свойство **Current** класса **XPathNodeIterator** возвращает объект **XPathNavigator**, который представляет текущий узел. Этот код просматривает набор узлов, отображая имя, тип и значение каждого узла:

```
while (iterator.MoveNext ())
{
    Console.WriteLine ("Type={0}, Name={1}, Value={2}",
        iterator.Current.NodeType,
        iterator.Current.Name,
        iterator.Current.Value);
}
```

Строка, возвращаемая свойством **Value** класса **XPathNavigator**, зависит от типа и содержимого узла. Так, если **Current** соответствует узлу атрибута или узлу элемента, содержащего просто текст (а не вложенные элементы), то **Value** возвращает значение атрибута или текстовое значение элемента. Если же **Current** соответствует узлу элемента, содержащему другие элементы, то **Value** возвращает тексты подэлементов, объединенные в одну длинную строку.

Каждый узел в наборе, возвращенном **Select**, может быть или одиночным узлом, или корнем дерева узлов. Просмотр дерева узлов, инкапсулированного **XPathNavigator**, немного отличается от просмотра дерева узлов в **XmlDocument**. Вот как происходит обход деревьев узлов, возвращенных **XPathNavigatorSelect**:

```
while ( iterator.MoveNext ())
    OutputNode (iterator.Current);

void OutputNode (XPathNavigator nav)
{
    Console. WriteLine ("Type={0}, Name={1}, Value={2}",
        nav.NodeType, nav.Name, nav.Value);
    if (nav.HasAttributes)
    {
        nav.MoveToFirstAttribute ();
        do
```

```

        {
            OutputNode (nav);
        } while (nav.MoveNextAttribute ());
        nav.MoveToParent ();
    }
    if (nav.HasChildren)
    {
        nav.MoveToFirstChild ();
        do
        {
            OutputNode (nav);
        } while (nav.MoveNext ());
        nav.MoveToParent ();
    }
}

```

**XPathNavigator** содержит семейство методов **Move**, позволяющих перемещаться по дереву узлов в любом направлении — вверх, вниз или в сторону. В данном примере используется пять таких методов:

**MoveToFirstAttribute**, **MoveToNextAttribute**, **MoveToParent**, **MoveToFirstChild** и **MoveToNext**. Заметьте также, что сам

**XPathNavigator** обеспечивает доступ к свойствам текущего узла аналогично **XmtTextReader**. Как вы могли бы использовать это в реальном приложении? Рассмотренное ранее приложение (проект XmlDocument) использует для извлечения содержимого XML-документа класс

**XmlDocument**. Для извлечения содержимого можно использовать и **XPath**, причем зачастую с меньшим объемом кода. Пример — приложение, код которого приведен ниже (проект XPathDemo) функционально эквивалентное рассмотренному ранее приложению (проект XmlDocument). Помимо демонстрации базовых приемов применения **XPathNavigator**, здесь также показано, что путем вызова **Select** для **XPathNavigator**, полученного через свойство **Current** итератора, вы можете выполнять подзапросы над наборами узлов, возвращенными запросами **XPath**. Сначала XPathDemo вызывает **Select** для создания набора узлов, представляющего все элементы *Car*, дочерние для *Cars*. Затем выполняется итерация по набору узлов, и для каждого узла *Car* вы-

**Программирование на языке C#. Урок 8.**

зывается **Select**, чтобы получить его дочерние элементы *Manufactured* и *Model*.

```
using System;
using System.Xml;
using System.Xml.XPath;
class MyApp
{
    static void Main ()
    {
        XPathDocument doc = new XPathDocument("../..//..//..//Cars.xml");
        XPathNavigator nav = doc.CreateNavigator();
        XPathNodeIterator iterator = nav.Select("/Cars/Car");
        while (iterator.MoveNext())
        {
            XPathNodeIterator it = iterator.Current.Select("Manufactured");
            it.MoveNext();
            string manufactured = it.Current.Value;
            it = iteratorF.Current.Select("Model");
            it.MoveNext();
            string model = it.Current.Value;
            Console.WriteLine("{0} {1}", manufactured, model);
        }
    }
}
```

## Использование XSL (XSLT)

**XSLT** (от **Extensible Stylesheet Language Transformations** — преобразования расширяемого языка таблиц стилей) — это язык преобразования документов из одного формата в другой. Хотя существует множество вариантов использования **XSLT**, основными являются:

- преобразование XML-документов в HTML-документы;
- преобразование XML-документов в другие XML-документы.

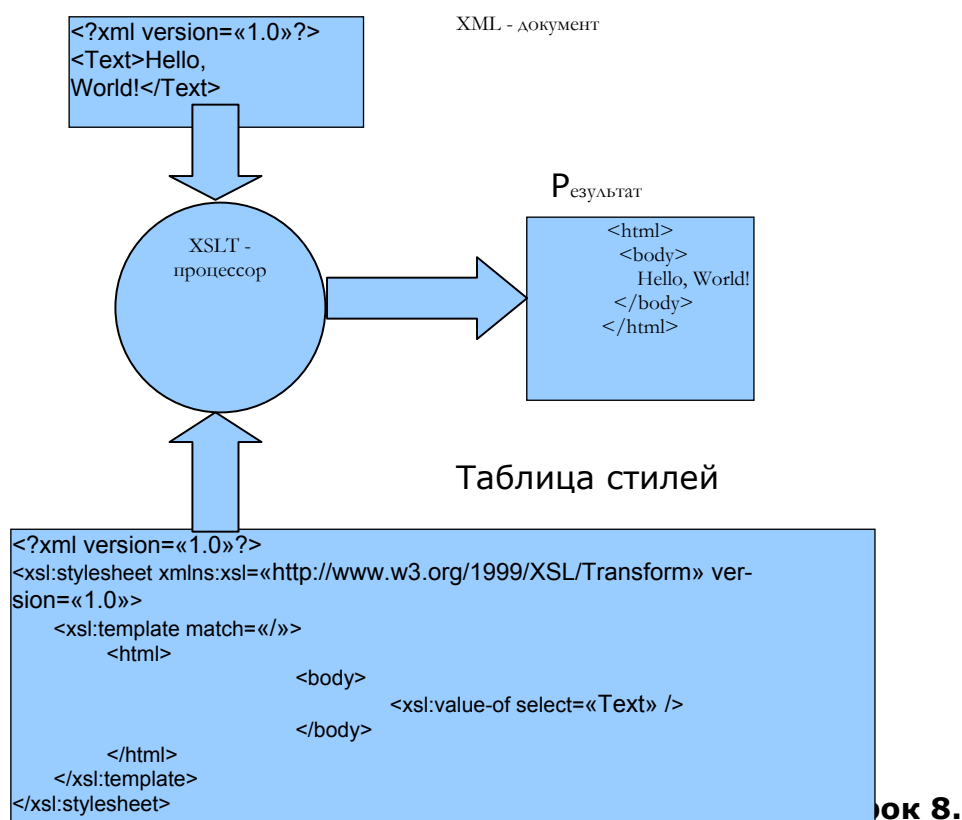
Первый вариант применяется для создания на XML Web-страниц и других документов для просмотра браузерами. XML определяет содержимое и структуру данных, но не определяет их представление. Применение

XSLT для генерации HTML из XML — прекрасный способ отделения содержимого от представления и создания универсальных документов, которые могут отображаться выбранным вами средством просмотра. Для наложения представления на XML-содержимое можно также использовать каскадные таблицы стилей (CSS), но **XSLT** мощнее CSS и предоставляет гораздо большие возможности по форматированию результатов.

**XSLT** также позволяет изменять формат XML-документов. Допустим, компания А ожидает, что счет-фактура от компании В должен соответствовать некоторому формату (т.е. некоторой схеме), но компания В уже использует другой формат счета-фактуры и не хочет менять его для удовлетворения прихотей компании А.

Тогда вместо того, чтобы отказаться от сотрудничества с компанией В, компания А может с помощью **XSLT** преобразовать формат счетов-фактур от В в свой формат. Так что обе компании удовлетворены, и никому не нужно напрягаться, чтобы обеспечить взаимодействие с партнером.

Механизм работы **XSLT** показан на рисунке ниже.



Вы задаете исходный документ (XML-документ, подлежащий преобразованию) и таблицу стилей **XSL**, определяющую преобразования, которые должен выполнить XSLT-процессор. Последний в свою очередь генерирует выходной документ по правилам, определенным таблицей стилей. **MSXML** — это **XSLT**-процессор. Таков же и класс **XslTransform**, расположенный в пространстве имен FCL **System.Xml.Xsl.XslTransform** — один из самых замечательных классов FCL, очень простой в использовании и превосходный инструмент для программного преобразования XML-документа в другой формат. Следующие разделы посвящены использованию **XslTransform**.

### Преобразование XML в HTML на клиенте

Если раньше вы никогда не работали с **XSLT**, то лучший способ познакомиться с ним — создать простой XML-документ и преобразовать его в HTML с помощью Internet Explorer (который для выполнения **XSL**-преобразований использует **MSXML**):

1. Скопируйте в выбранный вами каталог файлы Cars2.xml (приведен ниже)

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Cars.xsl"?>
<Cars>
  <Car>
    <Manufactured>La Hispano Suiza de Automovils</Manufactured>
    <Model>Alfonso</Model>
    <Year>1912</Year>
    <Color>Black</Color>
    <Speed>Rosewood</Speed>
  </Car>
  <Car>
    <Manufactured>BMW</Manufactured>
    <Model>Isetta</Model>
    <Year>1955</Year>
    <Color>Navy</Color>
    <Speed>85</Speed>
  </Car>
```

```
</Cars>
```

и Cars.xsl (приведен ниже)

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <body>
      <h1>MyCars</h1>
      <hr />
      <table width="100%" border="1">
        <tr bgcolor="gold">
          <td><b>Manufactured</b></td>
          <td><b>Model</b></td>
          <td><b>Year</b></td>
          <td><b>Color</b></td>
          <td><b>Speed</b></td>
        </tr>
        <xsl:for-each select="Cars/Car">
          <tr>
            <td><xsl:value-of select="Manufactured" /></td>
            <td><xsl:value-of select="Model" /></td>
            <td><xsl:value-of select="Year" /></td>
            <td><xsl:value-of select="Color" /></td>
            <td><xsl:value-of select="Speed" /></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

2. Временно удалите (или закомментируйте) следующий оператор в Cars.xml:

```
<?xml-stylesheet type="text/xsl" href="Cars.xsl"?>
```

3. Откройте Cars.xml в Internet Explorer; файл будет отображаться как XML



```

<?xml version="1.0" ?>
<!-- <?xml-stylesheet type="text/xsl" href="Cars.xsl"?> -->
- <Cars>
- <Car>
    <Manufactured>La Hispano Suiza de Automovils</Manufactured>
    <Model>Alfonso</Model>
    <Year>1912</Year>
    <Color>Black</Color>
    <Speed>Rosewood</Speed>
  </Car>
- <Car>
    <Manufactured>BMW</Manufactured>
    <Model>Isetta</Model>
    <Year>1955</Year>
    <Color>Navy</Color>
    <Speed>85</Speed>
  </Car>
</Cars>

```

4. Восстановите (или раскомментируйте) оператор, удаленный (или закомментированный) на шаге 2;

5. Откройте Cars.xml в Internet Explorer — на этот раз он будет отображен как HTML

## MyCars

Manufactured	Model	Year	Color	Speed
La Hispano Suiza de Automovils	Alfonso	1912	Black	Rosewood
BMW	Isetta	1955	Navy	85

Что же произошло? Оператор:

```
<?xml-stylesheet type="text/xsl" href="Cars.xsl"?>
```

является командой обработки, которая сообщает Internet Explorer, что Cars.xsl — это таблица стилей, содержащая инструкции для преобразования содержимого Cars.xml в другой формат. IE загружает таблицу стилей и применяет ее к Cars.xml, получая в результате HTML.

Большинство XSL-преобразований строится на основе шаблонов. Оператор:

```
<xsl:template match="/">
```

в Cars.xsl отмечает начало шаблона, применяемого ко всему документу. "/" - это выражение **XPath**, обозначающее корень документа. Первые несколько операторов шаблона выводят начало HTML-документа, содержащее HTML-таблицу.

Элемент *for-each* обрабатывает все элементы *Car*, дочерние для *Cars* (обратите на определяющее отбор выражение **XPath** «Cars/Car»), Для каждого элемента в таблицу добавляется одна строка, а элементы *value-of* инициализируют ячейки таблицы значениями соответствующих XML-элементов.

### Преобразование XML в HTML на сервере

Мы рассмотрели один способ преобразования XML в HTML. Его недостаток в том, что XML-документ должен содержать директиву обработки, указывающую таблицу стилей. Другой недостаток — выполнение преобразований на клиенте. Это не будет работать в IE 4, а может, и в большинстве браузеров других компаний, так как до последнего времени XSLT не был стандартизирован. Если только вы не в состоянии заставить клиентов использовать определенный тип браузера, вам следует выполнять преобразование на сервере, где можно быть уверенным в наличии соответствующего XSLT-процессора.

Как же выполнить XSLT-преобразование на сервере? Конечно, с помощью *XslTransform*. Пример демонстрирует страница Quotes.aspx (проект Quotes). На ней нет HTML — только серверный сценарий, генерирующий HTML из XML-документа Quotes.xml с помощью таблицы стилей Quotes.xsl. Преобразование выполняет метод *XslTransform.Transform*. Первый параметр — это объект *XPathDocument*, соответствующий исходному документу. Вторым (здесь он не используется) — задает *XsltArgumentList*, содержащий входные аргументы. Перед вызовом *XslTransform.Transform* Quotes.aspx вызывает другой метод *XslTransform* — *Load* — для загрузки таблицы стилей, определяющей преобразование. Полученная Web-страница показана на рисунке ниже.

## Лучшие цитаты

Цитата	Автор
Можно сопротивляться вторжению армий, вторжению идей сопротивляться невозможно.	В. Гюго
Страх - всегдашний спутник неправды.	В. Шекспир
Жалок тот ученик, который не превосходит своего учителя.	Леонардо да Винчи
Труд избавляет человека от трех главных зол - скуки, порока и нужды.	Ф. Вольтер
Нет ничего настолько исправного, чтобы в нем не было ошибок.	Ф. Петрарка

Как вы, вероятно, догадались, возможности XSLT гораздо шире, чем показывают эти примеры. Так, кроме элементов `for-each` и `value-of`, XSLT поддерживает элементы `if` и `choose` для условных переходов, элементы `variable` для объявления переменных, элементы `sort` для сортировки (`sort` используется и в `Quotes.xsl`) и др. Полный список элементов и сводку синтаксиса XSLT см. в спецификации XSLT (<http://www.w3.org/TR/xslt>).

Листинги с исходным кодом приведены ниже:

Файл `Quotes.aspx`

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Xml.XPath" %>
<%@ Import Namespace="System.Xml.Xsl" %>
<%
XPathDocument doc =
new XPathDocument (Server.MapPath ("Quotes.xml"));
XslTransform xsl = new XslTransform();
xsl.Load (Server.MapPath ("Quotes.xsl"));
xsl.Transform (doc, null, Response.OutputStream);
%>
```

Файл `Quotes.xml`

```
<?xml version="1.0"?>
<Quotes>
  <Quote>
    <Text>Жалок тот ученик, который не превосходит своего учителя.</Text>
    <Author>Леонардо да Винчи</Author>
  </Quote>
```

```

<Quote>
  <Text>Страх - всегдашний спутник неправды.</Text>
  <Author>В. Шекспир</Author>
</Quote>
<Quote>
  <Text>Можно сопротивляться вторжению армий, вторжению идей сопротивляться
невозможно.</Text>
  <Author>В. Гюго</Author>
</Quote>
<Quote>
  <Text>Нет ничего настолько исправного, чтобы в нем не было ошибок.</Text>
  <Author>Ф. Петрарка</Author>
</Quote>
<Quote>
  <Text>Труд избавляет человека от трех главных зол - скуки, порока и
нужды.</Text>
  <Author>Ф. Вольтер</Author>
</Quote>
</Quotes>

```

### Файл Quotes.xsl

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <body>
      <h1 style="background-color: teal; color: white;
font-size: 24pt; text-align: center; letter-spacing: 1.0em">
        Лучшие цитаты
      </h1>
      <table border="1">
        <tr style="font-size: 12pt; font-family: verdana; font-weight: bold">
          <td style="text-align: center">Цитата</td>
          <td style="text-align: center">Автор</td>
        </tr>
        <xsl:for-each select="Quotes/Quote">
          <xsl:sort select="Author" />
          <tr style="font-size: 10pt; font-family: verdana">
            <td><xsl:value-of select="Text" /></td>
            <td><i><xsl:value-of select="Author" /></i></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</template>

```

```
        </tr>
    </xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Проект находится в папке XML\_to\_HTML\_demo.

### **Домашнее задание**

1. С помощью класса XmlTextWriter напишите приложение, сохраняющее в xml-файл информацию о заказах. Каждый заказ представляет собой несколько товаров. Информацию, характеризующую заказы и товары разработать самостоятельно.
2. Используя XSLT- преобразование сгенерируйте HTML-документ из XML-документа, полученного в задании 1.
3. Прочитайте XML-документ, полученный в задании 1 с помощью классов XmlDocument и XmlTextReader и выведите полученную информацию на экран.