

Светлин Наков и колектив

ПРОГРАМИРАНЕ ЗА

.netTM Framework

1 ТОМ



Глава 12. Работа с XML

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания за езика XML и свързаните с него технологии

Съдържание

- Какво е XML?
- XML и HTML
- Кога се използва XML?
- Пространства от имена
- Схеми и валидация – DTD, XSD и XDR схеми
- Редакторът за схеми на VS.NET
- XML парсери
- XML поддръжка в .NET Framework
- Работа с DOM парсера – класовете **XmlNode** и **XmlDocument**
- SAX парсери и класът **XmlReader**
- Кога да използваме DOM и кога SAX?
- Създаване на XML документи с **XmlWriter**
- Валидация на XML по схема
- Работа с XPath – класовете **XPathNavigator** и **XPathDocument**
- XSL трансформации в .NET Framework

В тази тема...

В настоящата тема ще разгледаме работата с XML в .NET Framework. Ще обясним накратко какво представлява езикът XML. Ще обърнем внимание на приликите и разликите между него и HTML. Ще разгледаме какви са приложенията на XML. Ще се запознаем с пространствата от имена в XML и различните схеми за валидация на XML документи (DTD, XSD, XDR), като ще представим и средствата на Visual Studio .NET за работа с XSD схеми. Ще разгледаме особеностите на класическите XML парсери (DOM и SAX) и как те са имплементирани в .NET Framework. Ще опишем подробно класовете за работа с DOM парсера (**XmlNode** и **XmlDocument**) и ролята на класа **XmlReader** за SAX парсерите в .NET Framework. Ще опишем ситуациите, при които е подходяща употребата на DOM или SAX модела. Ще се

запознаем с начина на работа на класа **XmlWriter** за създаване на XML документи. Ще разгледаме начините за валидация на XML документи спрямо дадена схема с помощта на валидиращи парсери. Ще представим поддръжката в .NET Framework и на някои други XML-базирани технологии като XPath и XSLT.

Какво е XML?

Преди да преминем към класовете, които .NET Framework предоставя за работа с XML, нека първо разгледаме същността на тази технология.

XML (Extensible Markup Language)

XML първоначално е замислен като език за дефиниране на нови документни формати за World Wide Web. XML произлиза от SGML (Standard Generalized Markup Language) и на практика е негово подмножество със значително опростен синтаксис, което прави внедряването му много по-лесно. С течение на времето XML се налага като markup език за структурирана информация.

Какво представлява един markup език?

Произходът на термина **markup** е свързан с областта на печатните издания, но при електронните документи markup описва специфичното обозначаване на части от документите с тагове. Таговете имат две основни предназначения – те описват изгледа и форматирането на текста или определят структурата и значението му (метаинформация).

Днес се използват два основни класа markup езици – специализирани и с общо предназначение (generalized) markup езици. Първата група езици служат за генериране на код, който е специфичен за определено приложение или устройство и адресира точно определена необходимост. Общите markup езици описват структурата и значението на документа, без да налагат условия по какъв начин ще се използва това описание. Пример за специализиран markup език е HTML, докато SGML и неговото функционално подмножество XML са типични markup езици с общо предназначение.

XML markup – пример

Следният пример демонстрира концепцията на markup езиците с общо предназначение. При тях структурата на документа е ясно определена, таговете описват съдържанието си, а форматирането и представянето на документа не е засегнато – всяко приложение може да визуализира и обработва XML данните по подходящ за него начин.

```
<?xml version="1.0" encoding="windows-1251"?>
<messages>
  <message>XML markup описва структура и съдържание</message>
  <message>XML markup не описва форматиране</message>
</messages>
```

Универсална нотация за описание на структурирани данни

XML представлява набор от правила за съставяне на текстово-базирани формати, които улесняват структурирането на данни. Една от характеристиките, които налагат XML като универсален формат, е възможността да представя както структурирана, така и полуструктурирана информация. XML има отлична поддръжка на интернационализация, благодарение на съвместимостта си с Unicode стандарта. Друг универсален аспект на XML е способността му да отделя данните от начина им на представяне.

XML съдържа метайнформация за данните

XML спецификацията определя стандартен начин за добавяне на markup (метайнформация) към документите. Метайнформацията представлява информация за самата информация и нейната структура – така се осъществява връзката между данните, представени в XML документа, и тяхната семантика.

XML е метаезик

XML е метаезик за описание на markup езици. Той няма собствена семантика и не определя фиксирано множество от тагове. Разработчикът на едно XML приложение има свободата да дефинира подходящо за конкретната ситуация множество от XML елементи и евентуални структурни връзки между тях.

XML е световно утвърден стандарт

XML е световно утвърден стандарт, поддържан от W3C (World Wide Web Consortium, <http://www.w3.org>). XML не е самостоятелна технология, а по-скоро е основа на цяла фамилия от XML-базирани технологии като XPath, XPointer, XSLT и др., които също се поддържат и развиват от W3C.

XML е независим

Езикът XML е независим от платформата, езиците за програмиране и операционната система. Тази необвързаност го прави много полезен при нужда от взаимодействие между хетерогенни програмни платформи и/или операционни системи.

XML – пример

Следният кратък пример демонстрира един възможен начин за описание на книгите в една библиотека със средствата на XML. Информацията е лесно четима и разбираема, самодокументираща се и технологично-независима.

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

XML и HTML

Външно езикът XML прилича на езика HTML, но между двата езика има и сериозни различия.

Прилики между езиците XML и HTML

XML и HTML си приличат по това, че са текстово-базирани и използват тагове и атрибути.

Текстово-базирани

XML и HTML са текстово-базирани езици и това осигурява прозрачност на информационния формат. При нужда такива документи могат да се отворят и редактират с помощта на обикновен текстов редактор.

Използват тагове и атрибути

Двата езика използват елементи, всеки от които се състои от отварящ и затварящ таг (например `<book>` и `</book>`) и информация между тях (представяща съдържанието на елемента). Всеки елемент може да дефинира свои атрибути, които съдържат метаданни за съдържанието му.

Разлики между езиците XML и HTML

XML и HTML си приличат само външно. Те имат съвсем различно предназначение и това води до някои сериозни различия.

HTML е език, а XML – метаязык

Въпреки че и двата езика произлизат от SGML, на практика HTML е негово специализирано приложение, докато XML е функционално подмножество на SGML. HTML елементите и атрибутите са предефинирани и с ясно определен смисъл. XML от своя страна запазва гъвкавостта и разширяемостта на SGML - той не дефинира собствена семантика и набор от тагове, а предоставя синтаксис за описание на други езици.

HTML описва форматиране, а XML – структурирана информация

HTML е проектиран с единствената цел да осигури начин за форматиране на документи в World Wide Web. Той не е разширяем и не поддържа произволни структури от данни. За разлика от него XML предоставя средства за дефиниране на произволни тагове и структурни връзки между тях. HTML описва как да се представи информацията, докато XML описва самата информация, като я структурира по стандартен начин, разбираем за различни приложения.

HTML, XML и добре дефинираните документи

Въпреки че XML и HTML документите си приличат на външен вид (с тази разлика, че таговете на единия език са предефинирани, а на другия – не), XML синтаксисът е много по-строг и не допуска отклонения за разлика от HTML. В един HTML документ е допустима употребата на некоректно зададени тагове и те се игнорират впоследствие от браузъра, ако той не намери начин как да ги обработи. В XML спецификацията изрично се забранява на приложенията, обработващи XML документи, да гадаят смисъла на синтактично некоректен файл. Ако XML документът не е добре дефиниран, обработката му трябва да се прекрати и да се докладва за грешка.

Добре дефинирани документи

Езикът XML дефинира понятието "добре дефинирани документи" (well-formed documents). Да разгледаме какво точно означава това.

XML изисква добре дефинирани документи

Някои основни правила, които определят един XML документ като добре дефиниран, са следните:

- документът да има само един основен документен елемент
- таговете винаги да се затварят и то в правилен ред (да не се застъпват)
- атрибутите винаги да се затварят по правилен начин
- имената на таговете и атрибутите да отговарят на някои ограничения



Правете разлика между коренен ("/") и документен елемент (<library> в горния пример) в един XML документ. Тези понятия не са синоними!

Пример за лошо дефиниран XML документ

В следващия пример са нарушени почти всички правила, изброени по-горе – отварящи и затварящи тагове не си съответстват, тагове не се

затварят, не са спазени ограниченията за името на атрибута `bug!` и атрибутът `value` не е коректно затворен:

```
<xml>
  <button bug! value="OK name="b1">
    <animation source="demo1.avi"> 1 < 2 < 3
  </click-button>
< / xml >
```

Кога се използва XML?

Езикът XML има изключително широка употреба в съвременните софтуерни технологии, защото предоставя универсален формат за съхранение и обмен на информация, а от това имат нужда болшинството от съвременните софтуерни системи.

Обмяна на информация

Обмяната на информация между системи, които боравят с несъвместими формати, е сериозно предизвикателство в съвременното информационно общество. Много системи работят с нестандартизирани, собствени формати и при нужда от взаимодействие разработчиците трябва да полагат много усилия, за да осигурят съвместимост на обменяните данни при комуникацията.

XML е едно възможно решение на този проблем, тъй като позволява дефинирането на специфичен за приложението, прозрачен формат за обмяна на информация. XML документите, които спазват публикуваните от W3C спецификации, са разбираеми за всички приложения и така XML може да се използва като общ междинен формат при обмяната на информация.

Съхранение на структурирани данни

Почти всяко приложение има нужда от съхранение на данни. В много случаи XML е подходящ за тази задача, тъй като разделя структурираната информация от нейното визуално представяне. XML е подходящ формат за съхранение най-вече на малки информационни файлове или на данни, които не се очаква да поддържат произволно търсене (достъп). XML маркир описва структурата на данните наред с тяхното съдържание. Това позволява да се дефинират схеми за валидация на XML документи, чрез които да се установява валидността на XML структурата.

Недостатъци на XML

Наред с предимствата, които предоставя, XML понякога става причина за значително увеличение на размера на данните и времето, необходимо за обработката им.

Обемисти данни

XML е текстово-базиран формат, който използва тагове като ограничители (и описатели) на съдържаната в документа информация. Самата му природа (текстов формат с чести и повтарящи се етикети) е предпоставка за увеличен размер на файловете (съответно и увеличен мрежов трафик). Големината на един XML файл винаги е по-голяма от тази на файл със същата информация, записана в сравним двоичен формат.

Този недостатък обикновено може да бъде компенсиран на други нива. Дисковото пространство днес е далеч по-евтино, а алгоритмите за компресия позволяват бърза и качествена компресия при нужда (особено при текстови данни). Комуникационните протоколи като HTTP/1.1 могат да компресират информацията "в движение", спестявайки мрежов трафик толкова ефективно, колкото и при употребата на двоичен формат.

Повишена необходимост от физическа памет

Един XML документ може да бъде голям по размер по два критерия – в статичния си файлов формат (нужда от повече дисково пространство за съхранение) или в заредената в динамичната памет форма (нужда от повече изчислителни ресурси и RAM памет). Като пряко следствие от това, че XML данните са значителни по обем, идва повишената необходимост от физическа памет за съхраняването им.

Съвместимостта на XML с Unicode кодовата таблица също указва влияние. Например `short int` стойността 12355 има текстово представяне между 5 и 20 байта и само 2 байта, ако бъде съхранявана в двоична форма.

Намалена производителност

Заслужено или не, XML си е създал репутацията на технология, "лакома" за ресурси. XML се записва като текст и XML данните са в абстрактен логически формат, описващ тяхната структура. За прочитането им в едно приложение често са нужни две стъпки – парсване на XML информацията от нейния текстов вид и преобразуването на така получените данни, за да станат използвани от страна на приложението. Парсването и трансформацията изискват време, а същото важи и за генерирането на изходящ XML поток. Въпреки това, най-сериозната опасност за производителността идва от способността на XML да включва и зарежда външни ресурси (DTD файлове, XSD схеми).

Пространства от имена

Пространствата от имена представляват логически свързани множества, които изискват всички принадлежащи им имена да са уникални. Те служат за различаване на елементите и атрибутите от различни XML приложения, които притежават еднакви имена. Пространствата от имена групират всички свързани елементи и атрибути от едно XML приложение и улесняват разпознаването им от страна на софтуера.

Дефиниране на пространства от имена

Имената на елементите и атрибутите се състоят от две части – име на пространството, на което принадлежат, и локално име. Това съставно име е известно като квалифицирано име (qualified name, QName). Идентификаторите на пространствата от имена в XML трябва да се придържат към специфичен URI (Uniform Resource Identifier) синтаксис. URI спецификацията дефинира две основни URI форми – URL (Uniform Resource Locators) например `http://www.nakov.com/town` и URN (Uniform Resource Names) например `urn:nakov-com:country`.



URI идентификаторите на пространствата от имена не подлежат на анализ от страна на процесора – те са единствено средство за идентификация и няма изискване да сочат към реално достъпни ресурси в мрежата.

URI идентификаторите обикновено са доста дълги и вместо тях в XML документите се използва префикс за асоцииране на локалните елементи и атрибути с определено пространство от имена. Префиксът е просто съкратен псевдоним за един URI идентификатор, който се свързва с него при дефинирането на пространство от имена:

```
xmlns:<префикс>="<идентификатор на пространство от имена>"
```

Използване на тагове с еднакви имена – пример

Следващият пример демонстрира как пространствата от имена разрешават двусмислието при използване на тагове с еднакви имена в един XML документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<country:towns xmlns:country="urn:nakov-com:country"
  xmlns:town="http://www.nakov.com/town">
  <town:town>
    <town:name>Sofia</town:name>
    <town:population>1 200 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
  <town:town>
    <town:name>Plovdiv</town:name>
    <town:population>700 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
</country:towns>
```

Дефинирани са две пространства от имена: `urn:nakov-com:country` с префикс `country` и `http://www.nakov.com/town` с префикс `town`. Всяко от тези пространства съдържа елемент `<name>`, но проблем не съществува,

защото елементите са определени от префикса на собственото си пространство от имена – `<country:name>` описва името на държавата, докато `<town:name>` съдържа името на града.

Пространства по подразбиране

Използването на префиксно-ориентиран синтаксис е сравнително интуитивен процес за повечето софтуерни разработчици. Съществува обаче и друг начин за асоцииране на XML елементите с пространствата от имена – дефинирането на пространства по подразбиране. Използва се следният синтаксис:

```
xmlns="<идентификатор на пространство от имена>"
```

Пространствата по подразбиране не използват префикси. При дефиниране на такова пространство в един XML елемент всички неасоциирани с префикс (или друго пространство от имена) елементи в неговия обсег на видимост автоматично се свързват с пространството по подразбиране. Възможно е декларацията на пространство по подразбиране да бъде отменена – за целта на идентификатора му се присвоява празен низ:

```
<language xmlns="">C#</language>
```

Пространства по подразбиране – пример

Примерът демонстрира дефинирането на пространство по подразбиране <http://www.hranitelni-stoki.com/orders>. Елементът `<item>` не е изрично асоцииран с пространство от имена, затова той автоматично се свързва с пространството по подразбиране. Пълното име на елемента `<item>` е <http://www.hranitelni-stoki.com/orders:item>.

```
<?xml version="1.0" encoding="windows-1251"?>
<order xmlns="http://www.hranitelni-stoki.com/orders">
  <item>
    <name>бира "Загорка"</name>
    <amount>8</amount>
    <measure>бутилка</measure>
    <price>3.76</price>
  </item>
  <item>
    <name>кебапчета</name>
    <amount>12</amount>
    <measure>брой</measure>
    <price>4.20</price>
  </item>
</order>
```

Пространства от имена и пространства по подразбиране – пример

Следващият пример демонстрира един възможен начин на съвместна употреба на пространства по подразбиране и други пространства от имена:

```
<?xml version="1.0" encoding="utf-8" ?>
<faculty:student xmlns:faculty="urn:fmi"
  xmlns="urn:foo" id="235329">
  <name>Ivan Ivanov</name>
  <language xmlns="">C#</language>
  <rating>6.00</rating>
</faculty:student>
```

Дефинирани са пространство от имена `urn:fmi` с префикс `faculty` и пространство по подразбиране `urn:foo`. Елементът `student` принадлежи на пространството от имена с идентификатор `urn:fmi`, докато елементите `name` и `rating` са от пространството по подразбиране `urn:foo`. Елементът `language` от друга страна не принадлежи на нито едно пространство от имена, тъй като за него пространството по подразбиране е отменено.

В крайна сметка пълните имена на елементите от примерния документ са съответно `urn:fmi:student`, `urn:foo:name`, `urn:foo:rating` и `language`.



Автоматичното асоцииране с пространството по подразбиране на елементи, несвързани с друго пространство, не се отнася за атрибутите. Поради тази причина атрибутът `id` в горния пример не принадлежи на нито едно пространство от имена.

За разлика от пространството по подразбиране, префиксните пространства не могат да се отменят.

Схеми и валидация

Обичайно под думата схема се разбира общо представяне на даден клас предмети. В смисъла на XML, **схема** е формално описание на формата на XML документи.

Документ, който издържа успешно теста, описан от съответната XML схема, се определя като валиден (съобразяващ се със схемата). Процесът на тестване на документ спрямо зададена схема се нарича **валидация**.

Схемата гарантира, че документът изпълнява определени изисквания. Тя открива грешки в документа, които в последствие могат да доведат до неправилната му обработка. Схемите лесно се публикуват в Интернет и могат да служат като общодостъпен начин за описание на синтаксиса на дадено XML приложение.

XML схеми – защо са необходими?

Пространствата от имена дефинират синтаксис за групиране на свързани елементи от едно XML приложение и начин за обръщение към тях, но не разглеждат въпроса кои са тези елементи. Съдържанието на XML документите се контролира чрез дефиниране на схеми. Схемите контролират структурата на XML документите и дефинират необходимия синтаксис за целта. Схемите описват:

- допустими тагове, които могат да присъстват в един XML документ
- допустими атрибути за тези тагове
- допустими стойности за елементите и атрибутите в документа
- ред на поставянето на таговете в XML документа
- дефинират стойности по подразбиране

XML схеми – видове

Съществуват различни видове XML схеми, като всяка има своите силни и слаби страни. Ще разгледаме особеностите на няколко от най-популярните стандарти за XML схеми – DTD, XSD и XDR.

Езикът DTD

DTD (Document Type Definition) е формален език за описание структурата на XML документи. Този език е оригиналният XML документен модел, той присъства и в XML спецификацията. DTD всъщност датира отпреди времето на XML – DTD произтича от SGML стандартите, като основният синтаксис е запазен почти изцяло.

DTD съдържа правила за таговете и атрибутите в документа

DTD контролира структурата на един XML документ, като дефинира множество от разрешени за използване елементи. Други елементи извън описаните не могат да присъстват в документа.

Езикът дефинира модел на съдържание (content model) за всеки елемент. Този модел определя допустимите елементи или данни, които могат да се съдържат в един елемент, наредбата и броя им, а също и дали присъствието на определен елемент е задължително или избиращо.

DTD декларира множество от позволени атрибути за всеки елемент. Декларация на атрибут определя името, типа данни, стойностите по подразбиране (ако има такива) и указва дали атрибутът задължително трябва да присъства в документа или не.

Текстово-базиран език, но не е XML

DTD е текстово-базиран език, който е запазил в основна степен синтаксиса на SGML. DTD обаче не е XML-базиран стандарт – разработен е преди

появата на XML и днес малко по-малко отстъпва позициите си пред XML-базирани стандарти за схеми като XSD.

Дефиниране на DTD – пример

Следващият пример демонстрира една възможна DTD декларация, която контролира съдържанието на малка домашна библиотека:

<code>library.dtd</code>
<pre> <!-- contents of library.dtd --> <!ELEMENT library (book+)> <!ATTLIST library name CDATA #REQUIRED> <!ELEMENT book (title, author, isbn)> <!ELEMENT title (#PCDATA)> <!ELEMENT author (#PCDATA)> <!ELEMENT isbn (#PCDATA)> </pre>

Дефиниран е елемент с име `library`, който съдържа една или повече (но поне една) инстанция на елемента `book`. За елемента `library` е дефиниран списък от атрибути – `library` задължително трябва да притежава атрибут с име `name` от тип `CDATA` (*character data*).

Забележете, че DTD не определя `library` за документен елемент – DTD декларацията не може да разграничи никой елементите като кандидат за документен елемент в XML документа. Елементът `book` трябва да съдържа в себе си точно по една инстанция на елементите `title`, `author` и `isbn`. Те от своя страна са дефинирани като елементи от тип `PCDATA` (*parsed character data*).



Данните, дефинирани като CDATA (character data), не се обработват от парсера. Текстът в рамките на CDATA не се третира като markup, а се разглежда като чист текст. PCDATA (parsed character data) елементите подлежат на парсване и съдържанието им се третира като нормален markup.

Използване на DTD – пример

След като сме дефинирали горното DTD описание, лесно можем да го асоциираме с даден XML документ и след това да го използваме при валидация. За целта в този документ вмъкваме **ДОСТЪПЕ** декларация, която указва името на документния елемент (в този случай `library`) и относителния път до самата DTD декларация:

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE library PUBLIC "library.dtd">
<library name=".NET Developer's Library">
  <book>

```

```
<title>Programming Microsoft .NET</title>
<author>Jeff Prosise</author>
<isbn>0-7356-1376-1</isbn>
</book>
<book>
  <title>Microsoft .NET for Programmers</title>
  <author>Fergal Grimes</author>
  <isbn>1-930110-19-7</isbn>
</book>
</library>
```

Ключовата дума **PUBLIC** обозначава факта, че работим с публична външна DTD декларация (другата възможност е да се използва ключовата дума **SYSTEM** за частни DTD декларации).

XSD схеми

Въпреки че DTD присъства в оригиналната XML спецификация, с течение на времето става ясно, че е необходим по-мощен инструмент за описание на XML документите и това води до появата на езика XSD (XML Schema Definition Language).

Мощен XML-базиран език за описание на XML структурата

За разлика от DTD, XSD е XML-базиран език за описване структурата на XML документи. Той, подобно на DTD, съдържа съвкупност от правила за таговете в документа и техните атрибути. Езикът XSD осигурява система от типове за XML обработка, която е много по-силно типизирана, отколкото DTD.

Вградени типове данни

XML Schema предоставя набор от вградени типове данни, които разработчиците могат да използват, за да ограничават съдържанието на текста. Тези типове данни са описани в пространството от имена <http://www.w3.org/2001/XMLSchema>. Всеки от тях има дефинирана област от допустими стойности.

Потребителски-дефинирани типове данни

Към набора от предефинирани типове данни, XSD позволява употребата и на потребителски типове. XSD поддържа дефинирането на два основни потребителски класа – прости типове (чрез таг **xs:simpleType**, където **xs** е префикс за системното пространство от имена <http://www.w3.org/2001/XMLSchema>) и комплексни типове (чрез таг **xs:complexType**).

Простите типове не задават структура, а само стойностно поле, и могат да бъдат задавани само на текстови елементи (без наследници) и атрибути.

Елементите, притежаващи допълнителна структура – например с дефинирани атрибути или наследници – трябва да бъдат описани с комплексен тип.

Дефиниране на XSD схеми – пример

Настоящият пример демонстрира един възможен начин за дефиниране на XSD схема за валидация на съдържанието на малка домашна библиотека:

library.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="book" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
        use="optional" />
    </xs:complexType>
  </xs:element>
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title" />
        <xs:element ref="author" />
        <xs:element ref="isbn" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="title" type="xs:string" />
  <xs:element name="author" type="xs:string" />
  <xs:element name="isbn" type="xs:string" />
</xs:schema>
```

Схемата дефинира пет глобални елемента – **library**, **book**, **title**, **author** и **isbn** – като три от тях са дефинирани от тип **string** (**xs:string**, където **xs** отново е префикс за пространството от имена на **XMLSchema**), а **library** и **book** са дефинирани като комплексни типове. Всеки един от глобалните елементи може да бъде използван като документен елемент в XML файл. Структурата на **library** определя, че този елемент съдържа неограничен брой елементи **book** и има незадължителен атрибут **name** от тип **string**. Елементът **book** е съставен от **title**, **author** и **isbn** (точно в тази последователност) и не дефинира атрибути.

Използване на XSD схеми – пример

Гореописаната XSD схема лесно може да се асоциира с даден XML документи и да се използва за неговата валидация. Това става най-лесно

с помощта на дефинирания в пространството от имена `http://www.w3.org/2001/XMLSchema-instance` атрибут `noNamespaceSchemaLocation`, който указва относителния път до XSD документа, съдържащ съответната валидираща схема:

```
<?xml version="1.0" encoding="utf-8" ?>
<library name=".NET Developer's Library"
  xsi:noNamespaceSchemaLocation="library.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Освен `xsi:noNamespaceSchemaLocation` за адресиране на XSD схеми се използва и атрибутът `xsi:schemaLocation` – за схеми, свързани с определени пространства от имена.

XSD измества DTD

Един от най-големите недостатъци на DTD е липсата на поддръжка на пространства от имена, тъй като те са въведени по-късно. DTD изисква всеки елемент в един XML документ да има съответна декларация в DTD файла, което противоречи на идеята за XML пространствата от имена.

Друг съществен проблем пред DTD е слабо типизираната система от типове, която се прилага само за дефинираните атрибути. DTD е насочен главно към описание на структурата на един документ и обръща много по-малко внимание на съдържанието извън това дали елементите могат да съдържат character data или не. Единствено атрибутите могат да бъдат декларирани от различни типове (`ID`, `IDREF`, `enumerated`), но ограничения върху типа данни в един елемент не могат да бъдат налагани.

За справянето с такива проблеми е разработена нова система за описание на структурата и валидация на XML документите – XSD схеми. По-голямата изразителна мощ на XSD в сравнение с DTD декларациите води до постепенно налагане на XSD схемите като основно средство за валидация на документи.

XDR схеми

XDR (XML-Data Reduced) е още един XML-базиран език за описание на структурата на XML документи. Той е въведен от Microsoft преди появата

на стандартизираните от W3C XSD схеми. XDR е представен през 1999 като работна схема за валидация в продукта Microsoft BizTalk Server. XDR схемите са компактен вариант на XML-Data схемите. Те са по-мощни от DTD декларациите, но същевременно са по-слабо изразителни от XSD схемите, които се появяват през 2001.

В последно време XDR схемите губят своята популярност дори и при Microsoft-базираните продукти и технологии, където традиционно са намидали своето приложение (BizTalk, SQL Server 2000). XDR поддържа типове от данни и пространства от имена. Интересно е, че тези схеми могат да описват съответствия между структурата на XML документи и релационни бази данни.

Декларация на XDR схеми – пример

Следният пример демонстрира един възможен начин за дефиниране на XDR схема за валидация на малка домашна библиотека:

library.xdr

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="author" model="closed"
    content="textOnly" dt:type="string"/>
  <ElementType name="title" model="closed"
    content="textOnly" dt:type="string"/>
  <ElementType name="isbn" model="closed"
    content="textOnly" dt:type="string"/>
  <ElementType name="book" model="closed"
    content="eltOnly" order="seq">
    <element type="title" minOccurs="1" maxOccurs="1"/>
    <element type="author" minOccurs="1" maxOccurs="1"/>
    <element type="isbn" minOccurs="1" maxOccurs="1"/>
  </ElementType>
  <ElementType name="library" model="closed"
    content="eltOnly" order="seq">
    <AttributeType name="name" dt:type="string"
      required="yes"/>
    <attribute type="name"/>
    <AttributeType name="xmlns" dt:type="string"/>
    <attribute type="xmlns"/>
    <element type="book" minOccurs="1" maxOccurs="*" />
  </ElementType>
</Schema>
```

Схемата определя, че съдържанието на елементите `author`, `title` и `isbn` може да бъде единствено текст, но не и други елементи (`content="textOnly"`). Стойността на атрибута `model` (`closed`) показва, че тези елементи не могат да съдържат елементи и атрибути, освен изрично

споменатите в модела на съдържанието (content model) на съответния елемент (такива в този случай също няма).

Елементът `book` от своя страна не може да съдържа свободен текст, а само елементите, описани в неговия модел на съдържанието (`content="eltOnly"`), като те трябва да спазват точната последователност (`order="seq"`) – точно по един елемент в реда `title`, `author` и `isbn`.

Последният дефиниран елемент `library` може да съдържа неограничен брой елементи `book` (но най-малко един) и има задължителен атрибут `name` и незадължителен атрибут `xmlns`.



Ако декларацията за незадължителен атрибут `xmlns` не присъстваше, при определения `model="closed"` нямаше да е възможно да включваме други пространства от имена към XML документа, валидиран от тази XDR схема.

Друга възможност е да се използва `model="open"` и тогава елементи и атрибути, независимо че не са декларирани изрично в модела на съдържанието на даден елемент, могат да бъдат добавяни към него.

Използване на XDR схеми – пример

Така декларираната по-горе XDR схема можем без много усилия да приложим за валидация на XML документ. Необходимо е в документния му елемент да се съдържа специално форматиран атрибут за включване на пространство от имена:

```
<?xml version="1.0"?>
<library name=".NET Developer's Library"
  xmlns="x-schema:http://url-of-schema/library.xdr">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

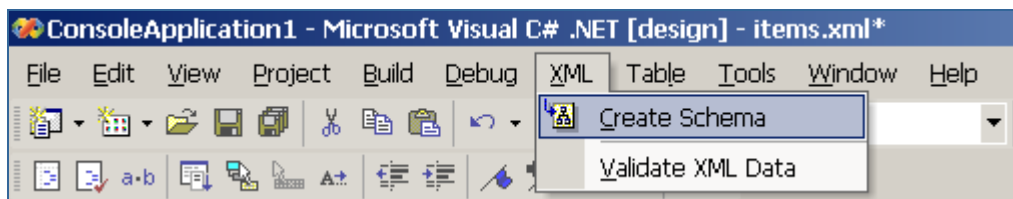
Когато XDR-съвместим парсер срещне пространство от имена, започващо с `x-schema`, той изтегля схемата от зададения URL адрес и извършва необходимата валидация.

Редакторът за схеми на VS.NET

VS.NET има силна поддръжка на XML и мощен редактор за XSD схеми. Нека разгледаме каква функционалност предоставя той.

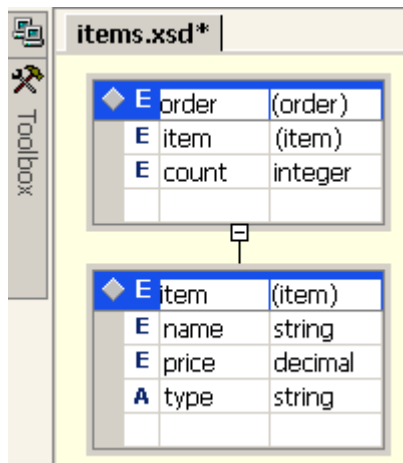
Поддръжка за XSD

Visual Studio .NET притежава вградена поддръжка за работа с XSD схеми. VS.NET позволява създаването на XSD схема по структурата на зададен XML документ (доколкото това е възможно). За целта при избран XML документ може да отидем в менюто XML и да изберем командата **Create Schema**. От същото меню може и да валидираме документ по вече създадена XSD схема с командата **Validate XML Data**.



Редактор за XSD схеми

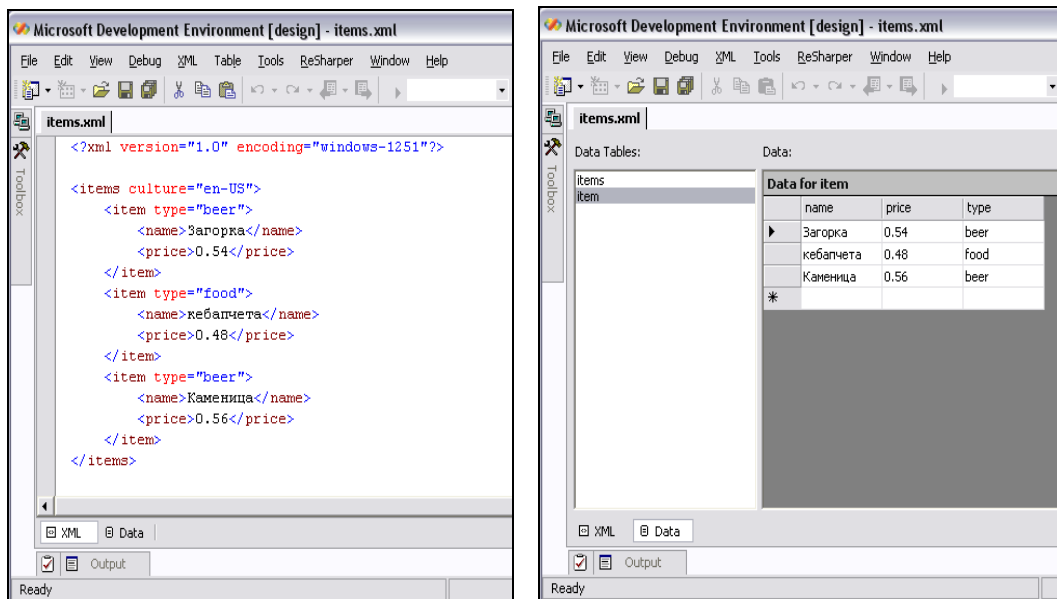
Visual Studio.NET разполага с вграден редактор за XSD схеми. Наред с възможността за промяна на XSD схемата в текстов режим, редакторът позволява и визуален режим на работа. Визуалният режим не е функционално ограничен – с него могат да се създават, променят и изтриват елементи, атрибути и типове, както и в текстовия режим на работа.



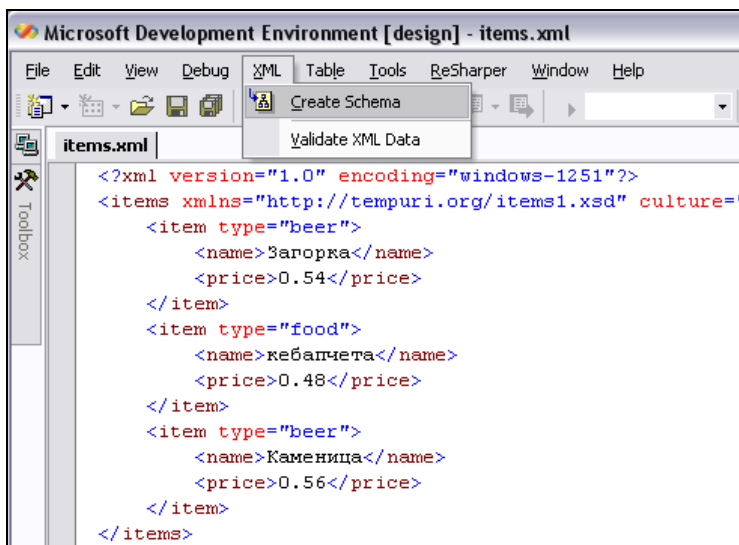
Създаване на XSD схема в VS.NET – пример

С настоящия пример ще демонстрираме как по визуален начин могат да се създават XSD схеми по наличен XML документ (в този случай `items.xml`). Ето и стъпките, които трябва да изпълним, за да получим желаната схема:

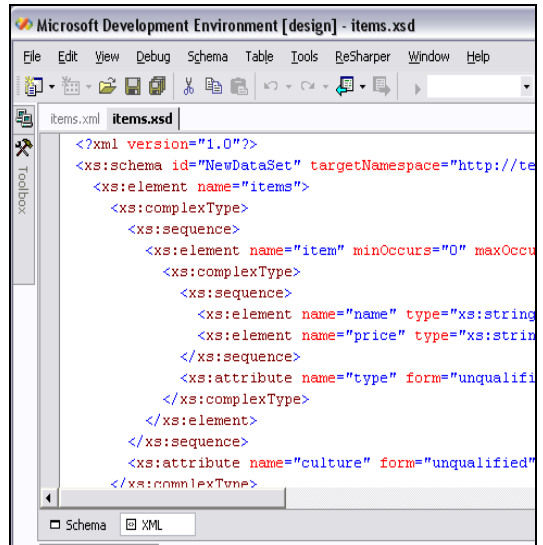
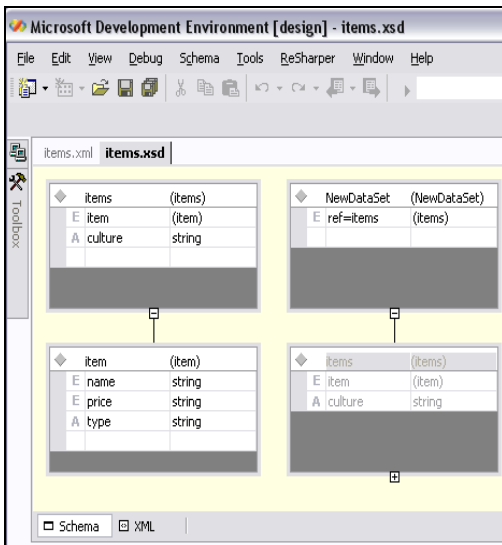
1. Стартираме VS.NET и отваряме XML документа `items.xml`. Файлът може да бъде разглеждан и редактиран освен като XML текст и като таблица с данни (смяната на двата режима правим от специален таб **"XML/Data"** в долната част на XSD редактора).



2. От менюто **"XML"** избираме **"Create schema"** и VS.NET създава XSD схема на базата на структурата на отворения XML документ.



3. Схемата, генерирана от VS.NET, можем да редактираме както като чист XML текст, така и чрез визуалния редактор за схеми (смяната на двата режима отново става от специален таб **"Schema/XML"** в долната част на XSD редактора).

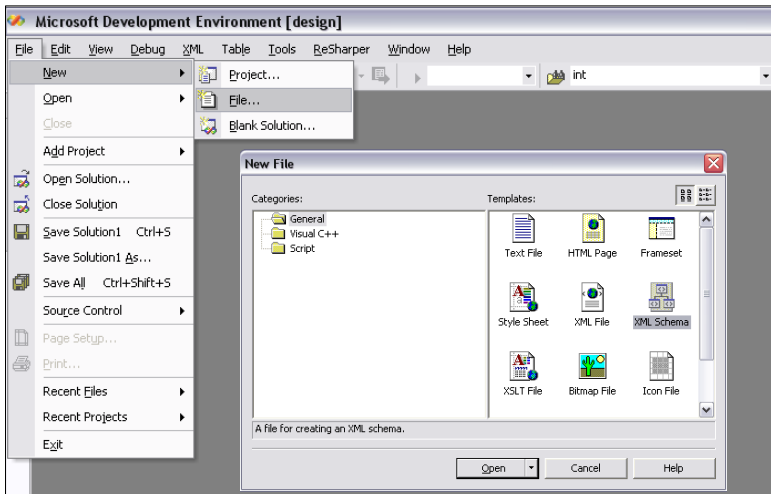


Редактиране на XSD схеми в VS.NET – пример

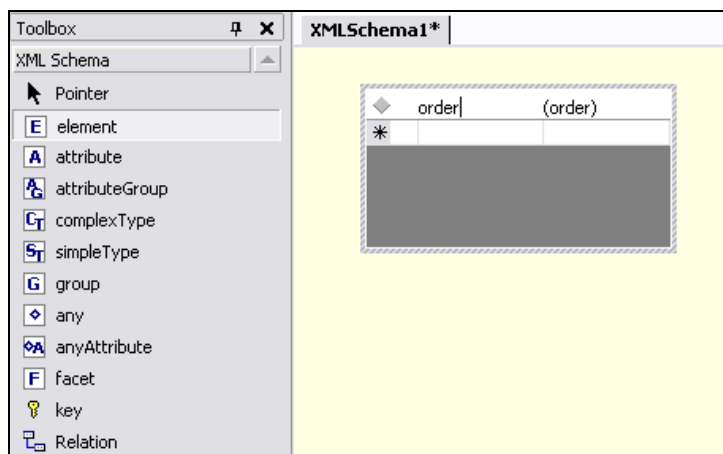
Сега ще демонстрираме как по визуален начин можем да редактираме XSD схеми във VS.NET. Ще създадем нова XSD схема, в която се дефинират три елемента – *order*, *items* и *item*. За всеки от тях визуално ще дефинираме необходимите елементи и атрибути и техните типове. Накрая отново във визуален режим на работа ще вградим елемента *item* в *items* и на свои ред *items* в *order*. Така със средствата на Visual Studio .NET ще създадем една примерна йерархична XSD схема, която можем да използваме за валидация на поръчки.

Нека проследим примера стъпка по стъпка:

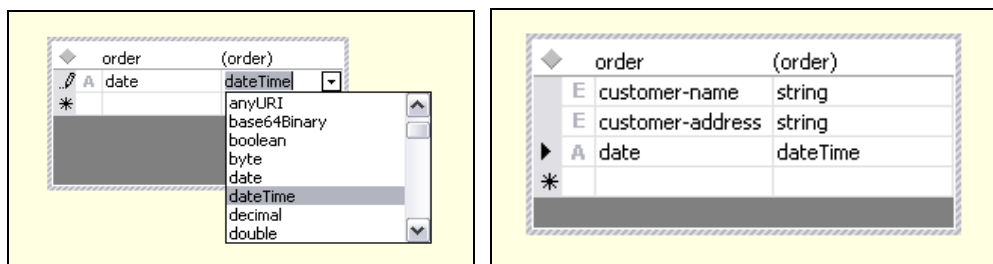
1. Избираме **File | New | File... | XML Schema** от менюто на VS.NET:



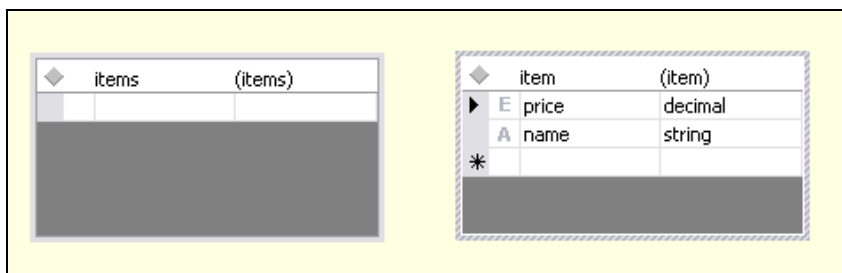
- От Toolbox прозореца на VS.NET доवличаме в схемата една контрола с име "element" и задаваме име на елемента "order".



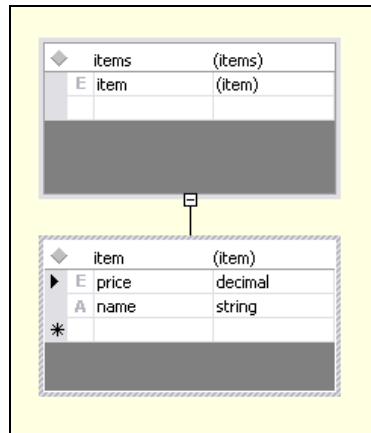
- Добавяме в елемента "order" атрибут "date" от тип "dateTime".
- Добавяме в елемента "order" елементи "customer-name" и "customer-address" от тип "string".



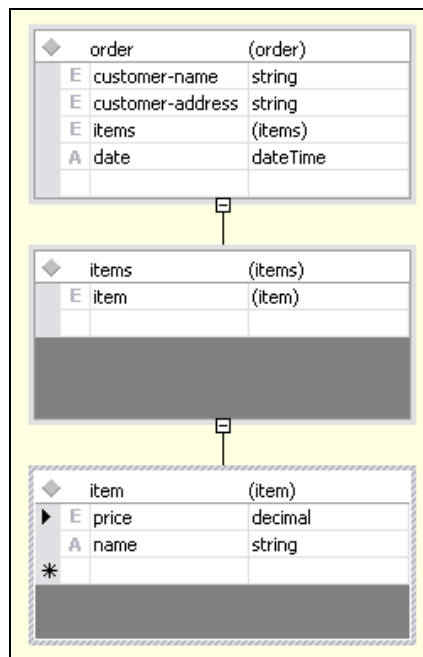
- От Toolbox на VS.NET довличаме в схемата два нови елемента и им задаваме имена съответно "items" и "item". В елемента "item" добавяме атрибут "name" от тип "string" и елемент "price" от тип "decimal".



- Довличаме елемента "item" върху елемента "items" и VS.NET автоматично влага в елемента "items" елемента "item".



7. Довличае елемента "item" върху елемента "items" и VS.NET автоматично влага в елемента "items" елемента "item".
8. Накрая доवличае елемента "items" върху елемента "order". VS.NET автоматично влага в елемента "order" елемента "items". В крайна сметка се получава така, че в елемента "order" е вложен елемента "items", а в него е вложен елементът "item".



XML парсери

Терминът **парсване** на един език се описва като процес на взимането на фрагмент код, описан в синтаксиса на този език, и разбиването му на отделни компоненти, дефинирани от езиковите правила. Понякога на

български език се използва терминът "синтактичен анализ", макар че той не винаги е точен превод на оригиналния английския термин **parse**.

XML парсерите са библиотеки, които четат XML документи, извличат от тях таговете и съдържаната в тях информация и ги предоставят за обработка на програмиста. Те предоставят и функционалност за построяване на нови и промяна на вече създадени XML документи.

XML парсери – приложение

XML парсерите предоставят стандартизиран интерфейс за някои основни операции, свързани с обработката на XML данни:

- извличане на данни от XML документи
- построяване на нови XML документи
- промяна на съществуващи XML документи
- валидация на XML документи по зададена схема

XML парсери – видове

XML парсерите могат да се класифицират по различни критерии. От една страна те се делят на валидиращи (нуждаят се от DTD или XSD схема, по която да валидират документите) и невалидиращи (изискват единствено добре дефинирани документи, които да обработват). По начина на работа се разграничават дървовидно-ориентирани (DOM, Document Object Model) и поточно-ориентирани (SAX, Simple API for XML Processing) парсери. Ще разгледаме накратко особеностите на последните два модела.

DOM

Ще започнем с DOM стандарта и ще разгледаме обектния модел, който той дефинира.

Какво представлява DOM?

Документният обектен модел DOM (Document Object Model) дефинира платформено и езиково-независим програмен интерфейс за достъп и манипулиране на съдържанието и структурата на XML документите като дървовидни структури в паметта. XML документният обектен модел е базиран на W3C DOM спецификацията и е утвърден световен стандарт. Той не е технология, специфична за .NET.

Документният обектен модел представя един XML документ като дървовидна йерархия от възли. DOM стандартът дефинира следните типове възли: **Document**, **Element**, **DocumentFragment**, **DocumentType**, **Attr**, **Text**, **EntityReference**, **ProcessingInstruction**, **Comment**, **CDATASection**, **Entity** и **Notation**. Някои от тези типове могат да имат наследници, като за всеки възел те са определени в DOM спецификацията. Документният обектен

модел определя също типовете `NodeList` за обработка на колекции и `NamedNodeMap` за речникови обекти от тип ключ-стойност.

DOM спецификацията описва интерфейси, а не действителни класове и обекти и затова за работа с нея ни е необходима конкретна имплементация (DOM парсер).

DOM приложения

DOM не е универсално решение, подходящо за всички случаи на обработка на XML документи. DOM обектната йерархия съхранява референции между различните възли в един документ. За целта целият XML документ трябва да е прочетен и парснат преди да бъде подаден на DOM приложението. При обработката на обемисти XML документи това може да се окаже сериозен проблем, защото е необходимо съхранението на целия документ в паметта. Въпреки това документният обектен модел е отлично решение в много ситуации. При нужда от произволен достъп до различни части от XML документа по различно време или за приложения, които променят структурата на XML документа "в движение", DOM предоставя отлична функционалност.

SAX

Сега нека сега разгледаме и SAX стандарта за обработка на XML документи и изясним кога е подходящо да се използва.

Какво представлява SAX?

SAX (Simple API for XML Processing) е базиран на събития, програмен интерфейс, който чете XML документи последователно като поток и позволява анализиране на съдържанието им.

Обработката на XML документи, базирана на събития, следи за наличието на ограничен брой събития, като срещане на отварящи и затварящи тагове на елементи, character data, коментари, инструкции за обработка и др. В процеса на прочитане на един документ SAX парсерът изпраща информация за документа в реално време чрез обратни извиквания. Всеки път, когато парсерът срещне отварящ или затварящ таг, character data или друго събитие, той известява за това програмата, която го използва.

SAX приложения

При SAX базираните приложения XML документът се предоставя за обработка на програмата фрагмент по фрагмент от началото до края. SAX приложението може да съхранява интересувашите го части, докато целият документ бъде прочетен, или може да обработва информацията в момента на получаването ѝ. Не е нужно обработката на вече прочетени елементи да чака прочитането на целия документ и още по-важно – не е нужно целият документ да се съхранява в паметта, за да е възможна работата с него. Тези характеристики правят SAX модела много удобен за обработка на обемисти XML документи, които не могат да бъдат заредени в паметта.

XML и .NET Framework

До момента направихме преглед на XML стандарта и по-важните технологии, свързани с него. Вече имаме стабилна основа, за да продължим с програмните средства, които .NET Framework предоставя за обработка на XML документи.

.NET притежава вградена XML поддръжка

За разлика от много програмни езици и платформи, който осигуряват средства за работа с XML под формата на добавки към основната функционалност, .NET Framework е проектиран от самото начало с идеята за силно интегрирана XML поддръжка.

Имплементациите на основните XML технологии се съдържат в асемблито **System.Xml**, където са дефинирани следните главни пространства от имена:

- **System.Xml** – осигурява основните входно-изходни операции с XML (**XmlReader** и **XmlWriter**), DOM поддръжка (**XmlNode** и наследниците му) и други XML помощни класове.
- **System.Xml.Schema** – осигурява поддръжка на валидация на XML съдържание чрез XML Schema (**XmlSchemaObject** и наследниците му).
- **System.Xml.XPath** – реализира функционалност за XPath търсене на информация и навигация в XML документ (класовете **XPathDocument**, **XPathNavigator** и **XPathExpression**).
- **System.Xml.Xsl** – предоставя възможност за XSL трансформации на XML документи (**XslTransform**).
- **System.Xml.Serialization** – осигурява сериализация до XML и SOAP (**XmlSerializer**).

.NET и DOM моделът

.NET Framework предоставя пълен набор от класове, които зареждат и редактират XML документи според W3C DOM спецификацията (нива 1 и 2). Основният XML DOM клас в .NET Framework е **XmlDocument**. Силно свързан с него е неговият клас-наследник **XMLDataDocument**, който разширява **XMLDocument** и акцентира върху съхраняването и извличането на структурирани таблични данни в XML.

При работа с XML DOM модела XML документът първо се зарежда целият като дърво в паметта и едва тогава се обработва. XML DOM предоставя средства за навигация и редактиране на XML документа и поддържа XPath заявки и XSL трансформации (ще ги разгледаме малко по-нататък).

Парсване на XML документ с DOM – пример

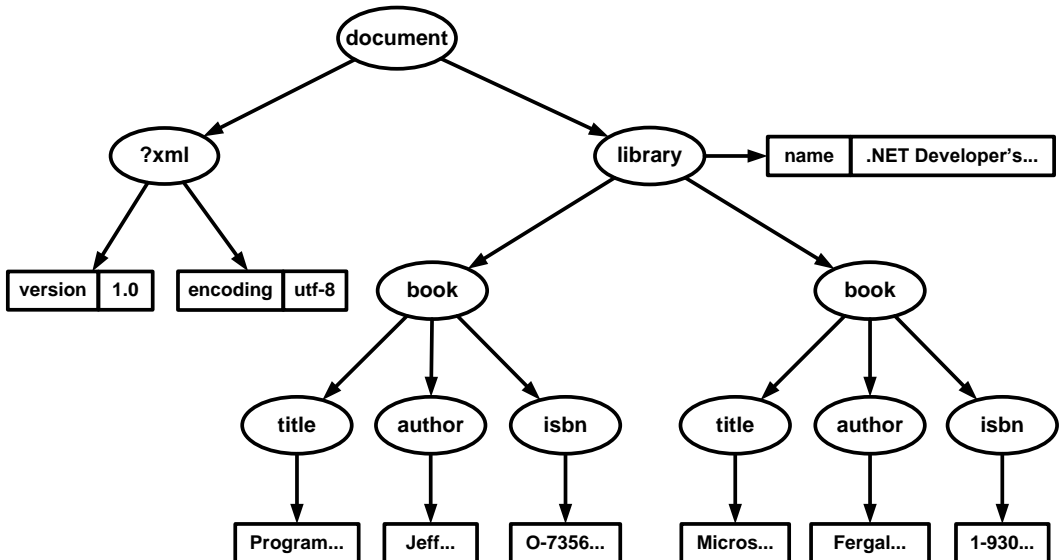
Преди да навлезем в детайлите на DOM парсера в .NET Framework, ще разгледаме кратък пример, който илюстрира използването му за парсване на XML документ, обхождане на полученото DOM дърво и извличане на информация от него.

За целта ни е необходим работен XML документ:

library.xml

```
<?xml version="1.0"?>
<library name=".NET Developer's Library"
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Този документ се представя в паметта като DOM дърво по следния начин:



Нашият пример има за цел да извлече книгите от файла `library.xml` и да отпечата информация за тях – заглавие, автор и ISBN. Ще го изградим стъпка по стъпка:

1. Създаваме нов проект във VS.NET.

2. Първото, което е необходимо да направим, е да заредим XML файла **library.xml**, за да можем в последствие да го подложим на обработка. След зареждането на документа отпечатваме съдържанието му, за да се уверим, че зареждането е успешно:

```
XmlDocument doc = new XmlDocument();
doc.Load("library.xml");
Console.WriteLine("Loaded XML document:");
Console.WriteLine(doc.OuterXml);
Console.WriteLine();
```

3. Извличаме документния елемент на XML файла и отпечатваме името му на конзолния изход:

```
XmlNode rootNode = doc.DocumentElement;
Console.WriteLine("Root node: {0}", rootNode.Name);
```

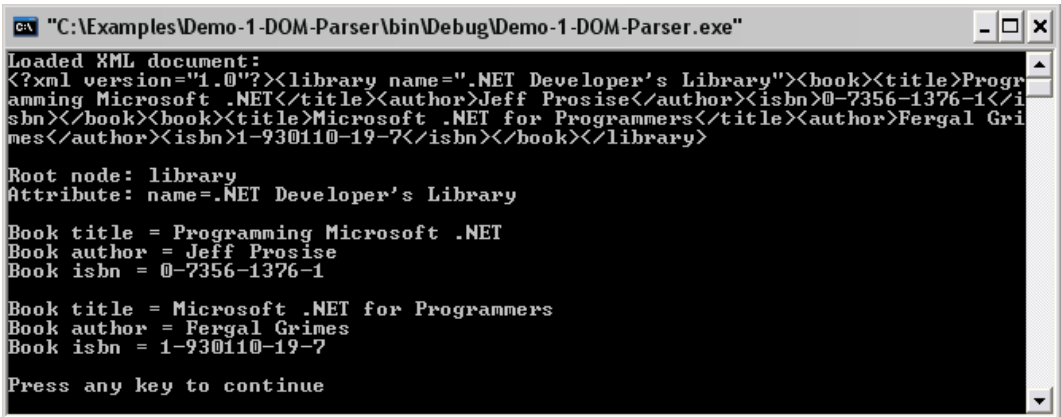
4. Обхождаме и отпечатваме атрибутите на документния елемент (в този случай имаме един единствен атрибут **name**):

```
foreach (XmlAttribute atr in rootNode.Attributes)
{
    Console.WriteLine("Attribute: {0}={1}",
        atr.Name, atr.Value);
}
```

5. Обхождаме всички елементи-деца на документния елемент. Всеки от тях описва една книга (елемент **book**). За всяка книга отпечатваме заглавието, автора и isbn номера, като ги извличаме от съответните им елементи, наследници на елемента **book**:

```
foreach (XmlNode node in rootNode.ChildNodes)
{
    Console.WriteLine("Book title = {0}",
        node["title"].InnerText);
    Console.WriteLine("Book author = {0}",
        node["author"].InnerText);
    Console.WriteLine("Book isbn = {0}",
        node["isbn"].InnerText);
    Console.WriteLine();
}
```

6. Това е всичко. Ето го резултатът, който получаваме на конзолния изход след компилацията и изпълнението на проекта:



```
C:\Examples\Demo-1-DOM-Parser\bin\Debug\Demo-1-DOM-Parser.exe
Loaded XML document:
<?xml version='1.0'?'><library name='.NET Developer's Library'><book><title>Programming Microsoft .NET</title><author>Jeff Prosise</author><isbn>0-7356-1376-1</isbn></book><book><title>Microsoft .NET for Programmers</title><author>Fergal Grimes</author><isbn>1-930110-19-7</isbn></book></library>

Root node: library
Attribute: name=.NET Developer's Library

Book title = Programming Microsoft .NET
Book author = Jeff Prosise
Book isbn = 0-7356-1376-1

Book title = Microsoft .NET for Programmers
Book author = Fergal Grimes
Book isbn = 1-930110-19-7

Press any key to continue
```

В примера използвахме класовете `XmlDocument`, `XmlNode` и `XmlAttribute`, от пространството `System.Xml`. Нека разгледаме за какво служат и как се използват тези класове.

Класовете за работа с DOM

Работата с DOM в .NET Framework се осъществява с помощта на следните по-важни класове:

- `XmlNode` – абстрактен базов клас за всички възли в едно DOM дърво.
- `XmlDocument` – съответства на корена на DOM дърво, обикновено съдържа два наследника: заглавна част (пролог) и документния елемент на XML документа.
- `XmlElement` – представя XML елемент.
- `XmlAttribute` – представя атрибут на XML елемент (двойка име-стойност).
- `XmlAttributeCollection` – списък от XML атрибути.
- `XmlNodeList` – списък от възли в DOM дърво.

Класът XmlNode

Да разгледаме най-важния клас в обектния модел на .NET за работа с XML – класът `XmlNode`.

Основополагащ при работата с DOM

Класът `XmlNode` е абстрактният клас, който представя възел в един XML документ. Той имплементира стандартизирания от W3C документен обектен модел (нива 1 и 2) и е ключът към работата с DOM в .NET Framework. Възли в един документ могат да бъдат елементи, атрибути, DOCTYPE декларации, коментари и дори целият XML документ.

XmlNode е базов клас за различните DOM типове възли

Класът **XmlNode** представя базов възел и е класът, наследяван от всички специфични DOM възли (**XmlDocument**, **XmlElement**, **XmlAttribute** и т.н.). Неговите свойства осигуряват достъп до вътрешните стойности на всеки възел: пространството от имена на възела, тип на възела, възел-родител, възел-наследник, съседни възли и др.

XmlNode позволява навигация в DOM дървото

Класът **XmlNode** предоставя набор от средства за навигация чрез своите свойства:

- **ParentNode** – връща възела-родител (или **null** ако няма).
- **PreviousSibling** / **NextSibling** – връща левия / десния съсед на текущия възел.
- **FirstChild** / **LastChild** – връща първия / последния наследник на текущия възел.
- **Item** (индексатор в C#) – връща наследник на текущия възел по името му.

XmlNode позволява работа с текущия възел в DOM дървото

- **Name** – връща името на възела (име на елемент, атрибут, ...).
- **Value** – връща стойността на възела.



Стойността на свойството Value в голяма степен зависи от типа на конкретно разглеждания възел. За възел от тип атрибут това свойство наистина връща стойността му, но за възел от тип елемент например, Value връща нулева референция. Стойността на елементите се достъпва през свойствата InnerText и InnerXml. За пълен списък на връщаните от Value стойности за различните DOM възли потърсете в MSDN.

- **Attributes** – връща списък от атрибутите на възела (като **XmlAttributeCollection**).
- **HasChildNodes** – връща булева стойност дали има възелът има наследници.
- **InnerXml**, **OuterXml** – връща частта от XML документа, която описва съдържанието на възела съответно без и с него самия.
- **InnerText** – връща конкатенация от стойностите на възела и наследниците му рекурсивно.
- **NodeType** – връща типа на възела (вж. изброения тип **XmlNodeType** в MSDN).

XmlNode позволява промяна на текущия възел

- **AppendChild(...)** / **PrependChild(...)** – добавя нов наследник след / преди всички други наследници на текущия възел.
- **InsertBefore(...)** / **InsertAfter(...)** – вмъква нов наследник преди / след указан наследник.
- **RemoveChild(...)** / **ReplaceChild(...)** – премахва / заменя указания наследник.
- **RemoveAll()** – изтрива всички наследници на текущия възел (атрибути, елементи, ...).
- **Value**, **InnerText**, **InnerXml** – променя стойността / текста / XML текста на възела.

Класът XmlDocument

Класът **XmlDocument** съдържа един XML документ във вид на DOM дърво според W3C спецификацията за документния обектен модел. Документът е представен като дърво от възли, които съхраняват елементите, атрибутите и техните стойности и съдържат информация за родител, наследник и съседни възли.

Да разгледаме неговите основни свойства, методи и събития

- **Load(...)**, **LoadXml(...)**, **Save(...)** – позволяват зареждане и съхранение на XML документи от и във файл, поток или символен низ
- **DocumentElement** – извлича документния елемент на XML документа.
- **PreserveWhitespace** – указва дали празното пространство да бъде запазено при зареждане / записване на документа.
- **CreateElement(...)**, **CreateAttribute(...)**, **CreateTextNode(...)** – създава нов XML елемент, атрибут или стойност на елемент.
- **NodeChanged**, **NodeInserted**, **NodeRemoved** – събития за следене за промени в документа.

Промяна на XML документ с DOM – пример

Ще разгледаме кратък пример, който демонстрира приложението на DOM парсера на .NET Framework за промяна на съдържанието на XML документ.

За работен XML документ ще използваме **items.xml**:

items.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
```



```
<name>Загорка</name>
<price>0.54</price>
</item>
<item type="food">
  <name>кебапчета</name>
  <price>0.48</price>
</item>
<item type="beer">
  <name>Каменица</name>
  <price>0.56</price>
</item>
</items>
```

Поставената задача е да удвоим цените на бирата в този XML документ, но същевременно да запазим непроменени цените на останалите стоки в списъка. За целта ще е необходимо да прочетем целия XML документ в паметта и да анализираме стоките една по една. При срещане на елемент, който идентифицираме като бира, удвояваме цената му, а в противен случай не предприемаме никакво действие.

Нека сега разгледаме стъпките за изграждане на приложението:

1. Стартираме VS.NET и създаваме нов проект – конзолно приложение.
2. Зареждаме работния XML документ `items.xml` в паметта, за да го подготвим за предстоящата манипулация:

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

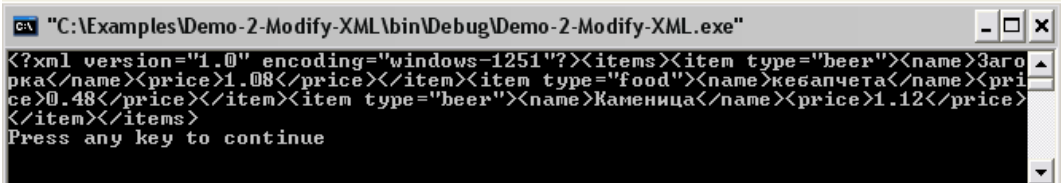
3. Естеството на този пример ни задължава да работим с десетични числа. В XML документа те са форматираны с десетична точка, но винаги съществува вероятност текущата активна култура на компютъра, където изпълняваме програмата, да е различна и да форматира числата с десетична запетая (например българската култура). За да се подсигури, че парсването на числата ще протече безпроблемно и няма да предизвика изключение от тип `FormatException`, най-правилно е да използваме специалната културно-необвързана култура, достъпна през свойството `CultureInfo.InvariantCulture`.
4. Обхождаме наследниците `item` на документния елемент `items` и за всеки от тях, чийто атрибут `type` има стойност `"beer"`, прочитаме стойността на наследника му `price`. Дотук обаче имаме стойността на елемента `price` единствено като низ. Необходимо е да парснем низа към десетично число и именно тук използваме `CultureInfo.InvariantCulture`. Вече разполагаме с десетично число, което удвояваме и записваме на мястото на старата стойност на елемента `price` (отново е нужно да укажем културата, за да се предпазим от грешки). Ето как изглежда кода, който извършва манипулацията:

```
foreach (XmlNode node in doc.DocumentElement)
{
    if (node.Attributes["type"].Value == "beer")
    {
        string currentPriceStr =
            node["price"].InnerText;
        decimal currentPrice = Decimal.Parse(
            currentPriceStr, CultureInfo.InvariantCulture);
        decimal newPrice = currentPrice * 2;
        node["price"].InnerText =
            newPrice.ToString(CultureInfo.InvariantCulture);
    }
}
```

5. Сега остава единствено да отпечатаме XML документа, за да се уверим, че промените действително са налице и след това да запазим промените в нов файл `itemsNew.xml`:

```
Console.WriteLine(doc.OuterXml);
doc.Save("itemsNew.xml");
```

6. Това е всичко. Ето резултата, който получаваме на конзолния изход след компилацията и изпълнението на проекта:



```
"C:\Examples\Demo-2-Modify-XML\bin\Debug\Demo-2-Modify-XML.exe"
<?xml version="1.0" encoding="windows-1251"?><items><item type="beer"><name>Загорка</name><price>1.08</price></item><item type="food"><name>кебапчета</name><price>0.48</price></item><item type="beer"><name>Каменица</name><price>1.12</price></item></items>
Press any key to continue
```

Можем да се уверим, че условието на задачата е изпълнено, като сравним двата XML документа `items.xml` и `itemsNew.xml`:



items.xml	itemsNew.xml
<code><?xml version="1.0" encoding="windows-1251"?></code>	<code><?xml version="1.0" encoding="windows-1251"?></code>
<code><items></code>	<code><items></code>
<code><item type="beer"></code>	<code><item type="beer"></code>
<code><name>Загорка</name></code>	<code><name>Загорка</name></code>
<code><price>0.54</price></code>	<code><price>1.08</price></code>
<code></item></code>	<code></item></code>
<code><item type="food"></code>	<code><item type="food"></code>
<code><name>кебапчета</name></code>	<code><name>кебапчета</name></code>
<code><price>0.48</price></code>	<code><price>0.48</price></code>
<code></item></code>	<code></item></code>
<code><item type="beer"></code>	<code><item type="beer"></code>
<code><name>Каменица</name></code>	<code><name>Каменица</name></code>
<code><price>0.56</price></code>	<code><price>1.12</price></code>
<code></item></code>	<code></item></code>
<code></items></code>	<code></items></code>

Построяване на XML документ с DOM – пример

За да илюстрираме по-пълно работата с DOM, ще разгледаме още един пример. Да си поставим за задача построяването на следния XML документ:

order.xml

```
<order>
  <item ammount="4">бира</item>
  <item ammount="2">картофки</item>
  <item ammount="6">кебапчета</item>
</order>
```

За целта трябва да създадем `XmlDocument`, да създадем и добавим документен елемент като негов наследник, след което да добавим към документния елемент още 3 елемента, като им зададем подходящо съдържание и им добавим по един атрибут за количество.

Ето примерна програма на C#, която реализира описаните стъпки:

```
using System.Xml;

class CreateXmlDemo
{
    static void AppendItem(XmlDocument aXmlDoc, XmlElement
        aXmlElement, string aItemName, int aAmmount)
    {
        XmlElement itemElement = aXmlDoc.CreateElement("item");
        itemElement.InnerText = aItemName;
        XmlAttribute ammountAttr =
            aXmlDoc.CreateAttribute("ammount");
        ammountAttr.Value = aAmmount.ToString();
        itemElement.Attributes.Append(ammountAttr);
        aXmlElement.AppendChild(itemElement);
    }

    static void Main()
    {
        XmlDocument xmlDoc = new XmlDocument();
        XmlElement docElement = xmlDoc.CreateElement("order");
        xmlDoc.AppendChild(docElement);
        AppendItem(xmlDoc, docElement, "бира", 4);
        AppendItem(xmlDoc, docElement, "картофки", 2);
        AppendItem(xmlDoc, docElement, "кебапчета", 6);
        xmlDoc.Save("order.xml");
    }
}
```

SAX парсери и XmlReader

В .NET Framework няма чиста имплементация на SAX парсер. Класът **XmlReader** има функционалност подобна на тази, предоставяна от класическите SAX парсери, но между тях има и определени разлики, които ще разгледаме по-подробно.

Класът XmlReader

XmlReader е абстрактен клас, който осигурява поточно-ориентиран едно-посочен достъп до XML данни само за четене. **XmlReader** е базиран на събития, както и SAX парсерите, но за разлика от тях е представител на pull модела, докато SAX парсерите по идея са push-ориентирани (двете понятия ще обясним малко по-надолу). Едно събитие указва начало или край на възел в процеса на прочитането му от потока от данни. В **XmlReader** информацията за настъпило събитие е достъпна през свойствата на класа, след като е извикан неговият **Read()** метод.

Разлика между pull и push парсер моделите

Съществуват два модела на работа на XML парсерите, обработващи поточно документи – push модел и pull модел.

Push парсер

Начинът на работа на push парсерите се характеризира с пряк контрол върху процеса на парсване, като настъпващите събития се предават **без изчакване** към клиентското приложение. Обикновено един push парсер изисква да се регистрира функция за обратно изискване (callback функция), която да обработва всяко събитие при настъпването му. Клиентското приложение не може да контролира парсването и трябва да съхранява информация за състоянието на парсера във всеки един момент, за да могат callback функциите да се изпълняват в правилен контекст (например трябва да помни колко дълбоко в XML дървото се намира в момента).

Pull парсер

При pull парсерите клиентското приложение упражнява активен контрол върху парсера. То изпълнява цикъл по събитията, идващи от парсера, като **изрично** извлича всяко следващо събитие. Приложението може да дефинира методи за обработката на специфични събития и изцяло да пропуска обработката на други, които не го интересуват. Това осигурява по-голяма ефикасност в сравнение с push модела, където всички данни задължително минават през клиентското приложение, защото само то може да прецени кои данни представляват интерес за него и кои – не.

XmlReader – основни методи и свойства

Нека сега направим преглед на най-важните методи и свойства на класа **XmlReader**:

- **Read()** – прочита следващия възел от XML документа или връща **false**, ако няма следващ
- **NodeType** – връща типа на прочетения възел
- **Name** – връща името на прочетения възел (име на елемент, на атрибут, ...)
- **HasValue** – връща дали възелът има стойност
- **Value** – връща стойността на възела
- **ReadElementString()** – прочита стойността (текста) от последния прочетен елемент
- **AttributeCount**, **GetAttribute(...)** – за извличане на атрибутите на XML елемент

Класът XmlReader – начин на употреба

XmlReader е абстрактен клас и осигурява само най-съществената функционалност за четене на XML документи. За работа с него се използват неговите наследници:

- **XmlTextReader** – за четене от файл или поток
- **XmlNodeReader** – за четене от възел в DOM дърво
- **XmlValidatingReader** – за валидация по XSD, DTD или XDR схема при четене от друг **XmlReader**

XmlReader – пример

Ще разгледаме кратък пример, който илюстрира работата с **XmlReader** за извличане на информация от XML документ.

Ще използваме следния работен документ:

library.xml

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
```

```
<title>Microsoft .NET for Programmers</title>
<author>Fergal Grimes</author>
<isbn>1-930110-19-7</isbn>
</book>
</library>
```

Целта на примера е да извлечем всички заглавия на книги, които се съдържат в XML документа и после да извлечем имената на всички елементи от документа.

Нека разгледаме необходимите стъпки:

1. Стартираме VS.NET и създаваме нов проект.
2. Отпечатваме на конзолния изход съобщение, че ще извличаме заглавията на книги от документа и инициализираме XML четеща:

```
Console.WriteLine("Book titles in the library:");
XmlTextReader reader = new XmlTextReader("library.xml");
```

3. Започваме да четем възлите един по един (при pull-базирания парсер **XmlReader** това означава, че изпълняваме цикъл по събитията, като с **Read()** извличаме всяко следващо събитие (т.е. възел). За всеки прочетен възел проверяваме дали е от тип елемент и дали името му съответства на търсените от нас заглавия на книги ("title"). В случай че възелът отговаря на тези условия, отпечатваме текстовата му стойност на конзолния изход:

```
while (reader.Read())
{
    if ((reader.NodeType == XmlNodeType.Element) &&
        (reader.Name == "title"))
    {
        Console.WriteLine(reader.ReadElementString());
    }
}
```

4. Дотук извлякохме всички заглавия на книги и ги отпечатахме. Сега остава да изпълним втората част от задачата – да отпечатаме имената на всички елементи от XML документа. Тъй като **XmlReader** е еднопосочен поточно-ориентиран парсер, необходимо е отново да го инициализираме преди да можем да го използваме. Извличането на имената на всички елементи е аналогично на описаното по-горе извличане на заглавията на книгите – отново влизаме в цикъл по събитията, като този път проверката се състои само в това да установи дали възлите са от тип елемент. При положение, че условието е изпълнено, отпечатваме името на текущия елемент на конзолния изход:

```
Console.WriteLine("\nElement names in the XML file:");
```

```

reader = new XmlTextReader("library.xml");
while (reader.Read())
{
    if (reader.NodeType == XmlNodeType.Element)
    {
        Console.WriteLine(reader.Name);
    }
}

```

5. Ето резултата, който получаваме на конзолния изход след като компилираме и изпълним програмата:

```

C:\Examples\Demo-3-XmlReader\bin\Debug\Demo-3-XmlReader.exe
Book titles in the library:
Programming Microsoft .NET
Microsoft .NET for Programmers

Element names in the XML file:
library
book
title
author
isbn
book
title
author
isbn
Press any key to continue

```

Кога да използваме DOM и SAX?

Моделът за обработка на XML документи DOM (`XmlDocument`) е подходящ, когато:

- обработваме малки по обем документи
- нуждаем се от гъвкавост при навигацията
- имаме нужда от пряк достъп до отделните възли на документа
- желаем да променяме документа

Моделът за обработка на XML документи SAX (`XmlReader`) е подходящ когато:

- обработваме големи по обем документи
- скоростта на обработка е важна
- не е необходимо да променяме възлите на документа

Класът `XmlWriter`

Класът `XmlWriter` осигурява бърз, еднопосочен, поточно-ориентиран способ за записване на XML данни във файлове и потоци. `XmlWriter` дефинира специални методи за записване различните съставни части на

един XML документ (елементи, атрибути, инструкции за обработка, коментари, и др.). XML писачът гарантира, че данните, които излизат от него, се съобразяват с W3C XML 1.0 стандарта и W3C спецификациите за пространства от имена.



XmlWriter генерира добре дефинирани документи, но само при положение, че потребителят подава коректна информация в процеса на създаване на нов XML документ.

XmlWriter не прави проверка за невалидни символи в имената на елементите и атрибутите. XML писачът не дава гаранции, че евентуална употреба на Unicode символи от страна на потребителя отговаря на текущата кодова таблица. В резултат на това символите без съответствие в кодовата таблица не се ескейпват и това може да доведе до некоректен изходен документ.

XmlWriter не проверява за дублирани имена на атрибути, нито валидира идентификаторите, задавани от потребителя при създаване на **ДОСТЪП** възел (например **SYSTEM** идентификатора).

XmlWriter не осигурява вградена валидация по схема или DTD декларация.

XmlWriter – основни методи

- **WriteStartDocument()** – добавя стандартна XML 1.0 пролог декларация в началото на документа (`<?xml ...`).
- **WriteStartElement(...)** – добавя отварящ таг за зададения елементен възел.
- **WriteEndElement()** – затваря най-вътрешния отворен елемент (използва кратък затварящ таг (`/>`), където е възможно).
- **WriteAttributeString(...)** – добавя атрибут в текущия елемент (методът добавя автоматично отварящи и затварящи кавички).
- **WriteElementString(...)** – добавя елемент по зададено име и текстова стойност.
- **WriteEndDocument()** – затваря всички отворени тагове и изпразва вътрешните буфери (чрез **Flush()**).

Работа с XmlWriter

XmlWriter е абстрактен клас, въпреки че някои от методите му имат конкретна имплементация. За работа с **XmlWriter** в .NET Framework се използва единственият му наследник – **XmlTextWriter**.

XmlTextWriter поддържа запис на XML данни в поток, файл или **TextWriter**. В конструктора му се задава необходимата кодираща схема

или се използва UTF-8 кодиране по подразбиране, ако не е определена схема. Класът осигурява стандартна имплементация на методите и свойствата на абстрактния **XmlWriter**, като към тях добавя и някои свои свойства:

- **Formatting** – избираме **Formatting.None**, ако XML данните не изискват отместване и **Formatting.Indented** в случай, че търсим подобрена четимост на документа (съдържанието на XML е идентично и при двата вида форматиране).
- **Indentation** – при избрана опция **Formatting.Indented**, **Indentation** определя броя символи, с които отмествае всяко следващо markup ниво.
- **IndentChar** – при избрана опция **Formatting.Indented**, **IndentChar** определя символа, който ще използваме за отместване на всяко следващо markup ниво. За да осигурим валидността на XML документа, трябва да използваме валидни XML символи за празно пространство.
- **Namespaces** – определя дали **XmlTextWriter** поддържа W3C XML пространства от имена.
- **QuoteChar** – определя какви кавички ще се използват при дефинирането на стойности на атрибути. **QuoteChar** може да бъде единична или двойна кавичка, всеки друг символ ще предизвика хвърляне на изключение от тип **ArgumentException**.

XmlWriter – пример

Ще разгледаме един кратък пример, който илюстрира работата с **XmlWriter** за създаване на нов XML документ. Целта на демонстрацията е да запишем информацията за няколко книги (заглавие, автор и isbn) от малка домашна библиотека в XML документ.

Нека разгледаме необходимите стъпки:

1. Стартираме VS.NET и създаваме нов проект
2. Създаваме нов обект от клас **XmlTextWriter**, като в конструктора подаваме името на изходния XML файл и избраната от нас кодиращата схема **windows-1251**:

```
XmlTextWriter writer = new XmlTextWriter("library.xml",
    Encoding.GetEncoding("windows-1251"));
```

3. За по-добра четимост на изходния документ указваме, че ще използваме една табулация като символ за отместване на всяко следващо markup ниво в документа:

```
writer.Formatting = Formatting.Indented;
writer.IndentChar = '\t';
```

```
writer.Indentation = 1;
```

4. Създаването на XML документа започва със записване на стандартен W3C XML 1.0 пролог в потока.

```
writer.WriteStartDocument();
```

5. Добавяме документен елемент с име **library** и дефинираме негов атрибут **name** със стойност **"My Library"**:

```
writer.WriteStartElement("library");  
writer.WriteAttributeString("name", "My Library");
```

6. За записването на книгите използваме помощна функция **WriteBook(...)**, която приема като параметри референция към **XmlWriter** обект и информацията за всяка книга (заглавие, автор, isbn). **WriteBook(...)** записва нов елемент **book** и в негови директни наследници **title**, **author** и **isbn** записва подадената в параметрите на функцията информация. Накрая затваряме елемента **book**:

```
private static void WriteBook(XmlWriter aWriter,  
    string aTitle, string aAuthor, string aIsbn)  
{  
    aWriter.WriteStartElement("book");  
    aWriter.WriteElementString("title", aTitle);  
    aWriter.WriteElementString("author", aAuthor);  
    aWriter.WriteElementString("isbn", aIsbn);  
    aWriter.WriteEndElement();  
}
```

7. С помощта на функцията **WriteBook(...)** добавяме в XML документа информацията за няколко книги:

```
WriteBook(writer, "Code Complete",  
    "Steve McConnell", "155-615-484-4");  
WriteBook(writer, "Интернет програмиране с Java",  
    "Светлин Наков", "954-775-305-3");  
WriteBook(writer, "Writing Solid Code",  
    "Steve Maguire", "155-615-551-4");
```

8. Сега остава единствено да затворим отворените тагове и да затворим **XmlTextWriter** обекта (така затваряме и потока, използван за писане във файла). Добре е тази операция да се извърши във **finally** клауза на **try-finally** блок (обхващащ записването на всички елементи в XML документа). По този начин сме сигурни, че и при непредвидени обстоятелства потокът не остава отворен:

```

try
{
    ...
    writer.WriteEndDocument();
}
finally
{
    writer.Close();
}

```

9. В крайна сметка получихме следната програма на C#:

```

using System;
using System.Xml;
using System.Text;

class XmlWriterDemo
{
    public static void Main()
    {
        XmlTextWriter writer = new XmlTextWriter("library.xml",
            Encoding.GetEncoding("windows-1251"));
        writer.Formatting = Formatting.Indented;
        writer.IndentChar = '\t';
        writer.Indentation = 1;
        try
        {
            writer.WriteStartDocument();
            writer.WriteStartElement("library");
            writer.WriteAttributeString("name", "My Library");
            WriteBook(writer, "Code Complete",
                "Steve McConnell", "155-615-484-4");
            WriteBook(writer, "Интернет програмиране с Java",
                "Светлин Наков", "954-775-305-3");
            WriteBook(writer, "Writing Solid Code",
                "Steve Maguire", "155-615-551-4");
            writer.WriteEndDocument();
        }
        finally
        {
            writer.Close();
        }
    }

    private static void WriteBook(XmlWriter aWriter,
        string aTitle, string aAuthor, string aIsbn)
    {
        aWriter.WriteStartElement("book");
        aWriter.WriteElementString("title", aTitle);
        aWriter.WriteElementString("author", aAuthor);
    }
}

```

```
aWriter.WriteElementString("isbn", aIsbn);  
aWriter.WriteEndElement();  
}  
  
}
```

10. Компилираме и изпълняваме програмата. Ето как изглежда съдържанието на генерирания XML документ `library.xml`, получен след компилацията и изпълнението на програмата:

```
library.xml  
  
<?xml version="1.0" encoding="windows-1251"?>  
<library name="My Library">  
  <book>  
    <title>Code Complete</title>  
    <author>Steve McConnell</author>  
    <isbn>155-615-484-4</isbn>  
  </book>  
  <book>  
    <title>Интернет програмиране с Java</title>  
    <author>Светлин Наков</author>  
    <isbn>954-775-305-3</isbn>  
  </book>  
  <book>  
    <title>Writing Solid Code</title>  
    <author>Steve Maguire</author>  
    <isbn>155-615-551-4</isbn>  
  </book>  
</library>
```

Валидация на XML по схема

Схемите (XSD, DTD и XDR) описват правила и ограничения за съставяне на XML документи. Те указват позволените тагове, реда и начините на влагането им, контролират позволените атрибути и техните стойности. Валидацията на XML документ по дадена схема в .NET Framework се извършва с помощта на валидиращи парсери.

Класът XmlValidatingReader

`XmlValidatingReader` е имплементация на абстрактния клас `XmlReader`, която осигурява поддръжка на основните схеми за валидация – XSD, DTD и XDR. `XmlValidatingReader` може да се използва за валидация както на цели XML документи, така и на XML фрагменти (т. е. без документен елемент). Този клас не може да бъде инстанциран директно от файл или URL адрес. Валидиращите четци в .NET Framework винаги работят върху

вече съществуващ XML четец и затова `XmlValidationReader` имплементира вътрешно само малка част от функционалността на родителския клас `XmlReader`.

XmlValidatingReader – основни методи, свойства и събития

- `Schemas` – връща `XmlSchemaCollection` обект, който съдържа колекция от предварително заредени XDR и XSD схеми. Предварителното зареждане ускорява процеса на валидация - схемите се кешират и няма нужда да се зареждат всеки път.



Предварителното зареждане и кеширане на схемите в `XmlValidatingReader.Schemas` е възможно за XSD и XDR схеми, но не и за DTD декларации.

- `SchemaType` (приложим само за XSD схеми) – връща съответния схема-обект за текущия възел на XML четеца в основата на `XmlValidatingReader`. По този обект можем да се ориентираме дали възелът е от вграден XSD тип, прост потребителски или комплексен потребителски тип.
- `ValidationType` – определя типа валидация, която ще се извършва. Възможните стойности (`Auto`, `None`, `DTD`, `XDR` и `Schema`) са дефинирани в изброения тип `ValidationType`.
- `XmlResolver` – определя `XmlResolver` обекта, който се използва за извличане на външни ресурси за схемите или DTD декларациите. `XmlResolver` се прилага и при обработване на `import` и `include` елементите, съдържащи се в XSD схемите.
- `Read()` – извикването на този метод премества XML четеца в основата на `XmlValidatingReader` към следващия възел от XML дървото. В същото време валидиращият четец взема информацията за възела и го валидира спрямо избраната схема и кешираната отпреди това информация.



`XmlValidatingReader` не предоставя метод, който валидира съдържанието на цял XML документ. Валидиращият четец обработва възлите един по един, така както така работи XML четецът в основата му.

- `ValidationEventHandler` – всяка грешка в процеса на валидация на възлите, води до възникване на събитието `ValidationEventHandler`. Методите, които обработват това събитие трябва да имат следната сигнатура:

```
public delegate void ValidationEventHandler(
```

```
object sender, ValidationEventArgs e);
```

Класът `ValidationEventArgs` съдържа символен низ `Message` с описание на грешката, `Exception` поле от тип `XmlSchemaException` съдържащо детайлно описание на грешката и `Severity` поле, което указва колко сериозен е проблемът.

Валидация на XML – пример

Ще разгледаме един кратък пример, който илюстрира валидацията на XML документ със средствата на класа `XmlValidatingReader`. Целта на примера е да валидираме готов XML документ по кеширана в паметта схема.

Нека се запознаем със съдържанието на XML документа, който ще валидираме:

library-valid.xml

```
<?xml version="1.0"?>
<library xmlns="http://www.nakov.com/schemas/library"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.nakov.com/schemas/library
    http://www.nakov.com/schemas/library.xsd"
  name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Указано е, че пространството от имена по подразбиране за този документ е `http://www.nakov.com/schemas/library` и схемата, която валидира това пространство е публикувана в Интернет на адрес `http://www.nakov.com/schemas/library.xsd`.

Ето и съдържанието на XSD схемата `library.xsd`:

library.xsd

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```

xmlns="http://www.nakov.com/schemas/library"
targetNamespace="http://www.nakov.com/schemas/library">

<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="book" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"
      use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="author"/>
      <xs:element ref="isbn"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="title" type="xs:string"/>
<xs:element name="author" type="xs:string"/>
<xs:element name="isbn" type="xs:string"/>

</xs:schema>

```

Ще резюмираме накратко XSD схемата – определят се пет глобални елемента – **library**, **book**, **title**, **author** и **isbn** – като последните три от тях са дефинирани от тип **string**, а **library** и **book** са дефинирани като комплексни типове. Структурата на **library** определя, че този елемент може да съдържа неограничен брой елементи **book** и има незадължителен атрибут **name** от тип **string**. Елементът **book** е съставен от елементи **title**, **author** и **isbn** и не дефинира атрибути. XSD схемата изрично указва, че описва елементите от пространството от имена **http://www.nakov.com/schemas/library**.

Да разгледаме процеса на валидация стъпка по стъпка:

1. Стартираме VS.NET и създаваме нов проект.
2. За да създадем валидиращ четец, необходимо е първо да инстанциираме обект от класа **XmlTextReader**, тъй като **XmlValidatingReader** не работи самостоятелно, а само върху вече създаден текстов четец:

```
XmlTextReader tr = new XmlTextReader("library-valid.xml");
```

```
XmlValidatingReader vr = new XmlValidatingReader(tr);
```

3. Дефинираме типа на предстоящата валидация и добавяме XSD схемата `library.xsd` в кеша на валидатора. Необходимо е изрично да укажем, че тя съответства на пространството от имена `http://www.nakov.com/schemas/library`. Ако не направим това, валидаторът ще търси схемата в Интернет от посочения в XML файла URL адрес `http://www.nakov.com/schemas/library.xsd`:

```
vr.Schemas.Add("http://www.nakov.com/schemas/library",  
    "library.xsd");  
vr.ValidationType = ValidationType.Schema;
```

4. Възможно е в процеса на валидация парсерът да открие невалиден таг, атрибут или друг проблем. Това води до възникване на събитието `ValidationEventHandler`. За да получим информация за конкретната грешка, закачаме метод-обработчик на това събитие с име `ValidationHandler(...)`. В него отпечатваме описанието и сериозността на проблема, като използваме съответните полета `Message` и `Severity` на класа `ValidationEventArgs` (параметър на метода-обработчик `ValidationHandler(...)`):

```
vr.ValidationEventHandler +=  
    new ValidationEventHandler(ValidationHandler);  
  
...  
  
public static void ValidationHandler(object sender,  
    ValidationEventArgs args)  
{  
    mValid = false;  
    Console.WriteLine("***Validation error");  
    Console.WriteLine("\tSeverity:{0}", args.Severity);  
    Console.WriteLine("\tMessage:{0}", args.Message);  
}
```

5. Задаваме стойност `true` на булевата член-променлива `mValid`. Тази променлива променяме единствено в метода-обработчик на събитието `ValidationEventHandler` т.е. ако XML документа е валиден, методът `ValidationHandler(...)` не се извиква нито веднъж и стойността на `mValid` остава `true`. Сега прочитаме целия XML документ възел по възел, което осигурява цялостната му валидация. Интересно е да отбележим, че не указваме изрично действие в процеса на валидация – при преминаването към всеки следващ възел, валидираният четец си взима необходимата информация и извършва проверката:


```
mValid = true;
while(vr.Read())
{
    // Do nothing, just read whole the document.
}
```

6. След като сме прочели целия документ, остава единствено да проверим стойността на булевата променлива `mValid` и ако тя е непроменена, отпечатваме съобщение за успешната валидация на конзолния изход:

```
if (mValid)
{
    Console.WriteLine("The document is valid.");
}
```

7. Ето как изглежда целия пример:

```
using System;
using System.Xml;
using System.Xml.Schema;

class XMLValidationDemo
{
    private static bool mValid;

    static void Main()
    {
        XmlTextReader tr = new XmlTextReader("library-valid.xml");

        XmlValidatingReader vr = new XmlValidatingReader(tr);

        vr.Schemas.Add("http://www.nakov.com/schemas/library",
            "library.xsd");
        vr.ValidationType = ValidationType.Schema;
        vr.ValidationEventHandler +=
            new ValidationEventHandler(ValidationHandler);

        mValid = true;
        while(vr.Read())
        {
            // Do nothing, just read whole the document.
        }

        if (mValid)
        {
            Console.WriteLine("The document is valid.");
        }
    }
}
```

```
public static void ValidationHandler(object sender,
    ValidationEventArgs args)
{
    mValid = false;
    Console.WriteLine("***Validation error");
    Console.WriteLine("\tSeverity:{0}", args.Severity);
    Console.WriteLine("\tMessage:{0}", args.Message);
}
}
```

8. Ето и резултата на конзолния изход, след като компилираме и изпълним програмата:



9. Нека сега вместо документа `library-valid.xml` да валидираме документа `library-invalid.xml`. Новият XML документ изглежда по следния начин:

`library-invalid.xml`

```
<?xml version="1.0"?>
<library xmlns="http://www.nakov.com/schemas/library"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.nakov.com/schemas/library
        http://www.nakov.com/schemas/library.xsd"
    name=".NET Developer's Library">
    <book>
        <title name="Programming Microsoft .NET" />
        <author>Jeff Prosise</author>
        <isbn>0-7356-1376-1</isbn>
    </book>
    <book>
        <book-title>Microsoft .NET for Programmers</book-title>
        <author>Fergal Grimes</author>
        <isbn>1-930110-19-7</isbn>
    </book>
</library>
```

Единствената промяна, която трябва да направим, е при създаването на `XmlTextReader` обекта в началото на програмата:

```
XmlTextReader tr = new XmlTextReader("library-invalid.xml");
```

10. Нека разгледаме съдържанието на конзолния изход след новата компилация и изпълнение на програмата:

```

C:\Examples\Demo-5-XML-Validation\bin\Debug\Demo-5-XML-Validation.exe
***Validation error
    Severity:Error
    Message:The 'name' attribute is not declared. An error occurred at file:
    ///C:/Examples/xml-files/library-invalid.xml, <7, 10>.
***Validation error
    Severity:Error
    Message:The element 'http://www.nakov.com/schemas/library:book' has inva
    lid child element 'http://www.nakov.com/schemas/library:book-title'. Expected 'h
    ttp://www.nakov.com/schemas/library:title'. An error occurred at file:///C:/Exam
    ples/xml-files/library-invalid.xml, <12, 4>.
***Validation error
    Severity:Error
    Message:The 'http://www.nakov.com/schemas/library:book-title' element is
    not declared. An error occurred at file:///C:/Examples/xml-files/library-invali
    d.xml, <12, 4>.
Press any key to continue_
  
```

Документът `library-invalid.xml` не е валиден XML документ, както можем сами да се уверим. Валидаторът открива три грешки – недеклариран атрибут `name` за елемента `title`, недеклариран елемент `book-title` и отново той не е валиден наследник на възела `book`.

Валидация на XML при DOM

Досега разгледахме процеса на валидация на XML документи с помощта на XML четци, но валидация може да се извършва и по време на конструирането на XML DOM дърво. Класът `XmlDocument` парсва цялото съдържание на подадения му XML документ в паметта чрез метода `Load(...)`. Този метод прави проверка единствено дали XML е добре дефиниран, но не го валидира спрямо схема или DTD декларация.

За да валидираме DOM дървото в процеса на изграждането му е необходимо да използваме специален конструктор на метода `Load(...)`:

```
public override void Load(XmlReader);
```

Едно XML DOM дърво може да бъде създадено по различни източници включително поток, текстов четец и име на файл. Ако заредим документа през `XmlValidatingReader` (наследник на `XmlReader`), постигаме валидация на DOM дървото едновременно с неговото изграждане. Ще скицираме необходимия за целта сорс код:

```

XmlDocument doc = new XmlDocument();
XmlTextReader tr = new XmlTextReader("Sample.xml");
XmlValidatingReader valReader = new XmlValidatingReader(tr);
valReader.ValidationType = ... ;
valReader.ValidationEventHandler += ... ;
doc.Load(valReader);
  
```

XPath

До момента разгледахме доста неща, свързани с работата с XML, но това не е всичко. XML технологиите са много и ролята им в съвременното програмиране постоянно нараства. Затова ще разгледаме още няколко от тези технологии – XPath и XSLT.

Ще започнем от технологията XPath, която има широко приложение при извличане на информация от XML документи и се използва като съставна част в други XML технологии.

Описание на езика

XPath спецификацията се появява скоро след публикуването на XML 1.0 стандарта и представлява W3C утвърден език за адресиране на части от XML документи. XPath изразите приличат на описания на пътища от файловата система, съставени от имена на файлове и директории (оттам идва и името на езика XPath).



Езикът XPath обслужва XML документи, но синтаксисът му не е XML-базиран.

Ще дадем много кратко описание на езика XPath без да навлизаме в подробности, след което ще дадем няколко примера.

Какво представлява един XPath израз?

XPath изразите съдържат описания на пътища до възли и критерии, на които тези възли трябва да отговарят. Описанията могат да бъдат относителни или абсолютни и това определя в какъв контекст се оценява една XPath заявка.



Един XPath израз винаги се оценява в определен контекст (контекстен възел, контекстно множество от възли).

В началото контекстният възел се определя от приложението и представлява начална точка за XPath заявката. На всяка стъпка от описания XPath път контекстният възел приема стойността на текущия възел. Възлите, които имат отношение към частта на XPath заявката, изпълнявана в момента, образуват контекстно множество. Множеството възли, което се връща като краен резултат, включва само част от тези възли, които отговарят на зададени допълнителни условия.

XPath стъпка

Един XPath път се състои от една или повече **локационни стъпки** (разделени със символ "/"). Всяка стъпка се състои от ос (незадължителен елемент), тест на възли и предикат (също незадължителен елемент):

```
ос::тест-на-възли[предикат]
```

Ос на XPath стъпката

Оста определя йерархичната връзка между контекстния възел и възлите, избирани на една локационна стъпка (т. е. определя контекстното множество възли за всяка стъпка). Ако XPath стъпка няма зададена ос, по подразбиране в контекстното множество възли участват преките наследници на контекстния възел. Нека да разгледаме някои възможни стойности за ос на XPath стъпка:

- **self(.)** – включва самия контекстен възел
- **child** – включва преките наследници на контекстния възел
- **parent(..)** – включва родителят на контекстния възел
- **descendant** – включва възлите от поддървото с корен контекстния възел (участват само възлите от тип елемент, текст и инструкция за обработка, но не и коментари или атрибути)
- **descendant-or-self(//)** – разновидност на **descendant**, която включва към дървото и самия контекстен възел
- **ancestor** – включва предшествениците на контекстния възел в йерархията чак до коренния елемент
- **ancestor-or-self** – разновидност на **ancestor**, включва към множество и самия контекстен възел
- **attribute(@)** – включва атрибутите на контекстния възел, ако той е от тип елемент



При използване на съкратената форма за осите (указана в скоби по-горе) не се използва разделител "::" между оста и теста на възли.

Тест на възли

Тестът на възли е основан на възли израз, който се оценява за всеки възел в контекстното множество. Ако тестът върне положителен резултат, възелът остава в множеството, а в противен случай се премахва оттам. Обикновено тестът на възли се състои от описание на път до даден възел и връща положителен резултат, ако пътят съществува в текущия контекст. Той може да съдържа XPath функции:

- **text()** – връща текстовото съдържание на контекстния възел
- **comment()** – връща всички наследници на контекстния възел от тип коментар

- `processing instruction()` – връща всички наследници на контекстния възел, които са от тип инструкция за обработка
- `node()` – връща всички наследници на контекстния възел

Предикат

Предикатът е незадължителен логически израз, който се прилага като допълнителен филтър върху текущото множество от възли, получено след изпълнението на възловия тест. Той също може да съдържа XPath функции (например `count(...)` връща броя на възлите в множеството подадено като параметър). Една XPath стъпка може да има повече от един предикат, като те се записват един след друг подобно на индекси на многомерен масив.

XPath изразите в действие

Нека разгледаме няколко практически XPath примери, които ще ни помогнат да разберем начина на работа на езика XPath:

- `/` – адресира коренния елемент на документа
- `/someNode` – адресира всички възли с име `someNode`, преки наследници на корена
- `/books/book` – адресира всички възли `book`, наследници на възел `books` (`books` от своя страна е пряк наследник на корена)
- `books/book` – адресира всички възли `book`, наследници на възел `books` (няма ограничения за местоположението в документа на елемента `books`)
- `/books/book[price<"10"]/author` – адресира всички автори (`/books/book/author`), чиито книги имат цена по-малка от 10
- `/items/item[@type="food"]` – адресира всички възли с име `item`, които имат атрибут `type` със стойност `"food"` и са наследници на възел `items`, пряк наследник на корена на документа

Сега ще демонстрираме един по сложен пример за финал, като ще опишем процеса на адресация на възлите стъпка по стъпка – `/book/chapter[3]/para[last()][@lines > 10]`:

1. Адресираме елементите `book`, преки наследници на корена на документа.
2. Измежду възлите-наследници на `book` с име `chapter` адресираме само третите по ред.
3. Измежду техните възли-наследници `para` адресираме само последните по ред с това име, които имат атрибут `lines` със стойност по-голяма от 10.

XPath, XSLT и XPointer

XPath често пъти се интегрира с XSLT и XPointer. Езикът XPath обикновено се използва за претърсване на XML DOM източник на данни и служи за филтриране на възлите, върху които се прилага определена XSL трансформация. XPath се налага все повече и при XPointer – формализъм за идентифициране на фрагменти от XML документи. Ето един пример за интеграция на XPath и XPointer:

```
library.xml#xpointer (/library/book[isbn='1-930110-19-7'])
```

Този израз сочи към възел `book`, наследник на `library`, който има наследник `isbn` със стойност `'1-930110-19-7'`.

XPath в .NET Framework

.NET Framework осигурява пълна поддръжка за езика XPath чрез класовете в пространството от имена `System.Xml.XPath`. Имплементацията на XPath е основана на езиков парсер и оценяващ модул. Общата архитектура прилича на тази при заявките към бази данни – както SQL командите, XPath изразите се подготвят предварително и се подават на оценяващ модул по време на изпълнение на програмата.

XPath и XmlNode

Средствата за работа с XPath в .NET Framework, освен през класовете от пространството `System.Xml.XPath` (програмен интерфейс, основан на концепцията за XPath навигатор), са достъпни директно през XML DOM модела (класът `XmlNode`).

Извикването на XPath изрази през класа `XmlNode` винаги се извършва в контекст на вече съществуваща инстанция на класа `XmlDocument`. Програмният интерфейс на този подход носи особеностите на стария COM-базиран MSXML стил на програмиране, популярен доскоро при Win32 приложенията за работа с XML. XML DOM поддръжката за XPath изрази улеснява преминаването от MSXML към .NET Framework стила и предоставя вграден механизъм за търсене на възли в зареден в паметта XML документ.

XPath и XmlNode – методи

XPath може да се използва директно от класа `XmlNode` и всички негови наследници през следните методи:

- `SelectNodes(string xpathQuery)` – връща списък от всички възли, които съответстват на зададения XPath израз.
- `SelectSingleNode(string xpathQuery)` – връща първия възел, който съответства на зададения XPath израз.

XPath и XmlNode – пример

Следният кратък пример илюстрира работата с XPath и `XmlNode` за търсене на възли в един XML документ.

Ще използваме следния работен документ `items.xml`:

`items.xml`

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Целта на демонстрацията е да открием имената на всички стоки от тип **"beer"** в този документ.

Нека разгледаме необходимите стъпки, за да постигнем това:

1. Стартираме VS.NET и създаваме нов проект.
2. Както споменахме вече, съвместната работа на XPath и `XmlNode` изисква съществуваща инстанция на `XmlDocument`. Затова първо създаваме нов `XmlDocument` и зареждане в него XML документа `items.xml`:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("items.xml");
```

3. Сега вече можем да подготвим самия XPath израз, който описва всички имена на стоки от тип бира в зададения XML документ - търсим всички възли `name`, наследници на тези възли `item`, чиито атрибут `type` има стойност **"beer"**. Възлите `item` от своя страна трябва да са преки наследници на възли `items`, закачени за корена на документа:

```
string xpathQuery = "/items/item[@type='beer']/name";
```

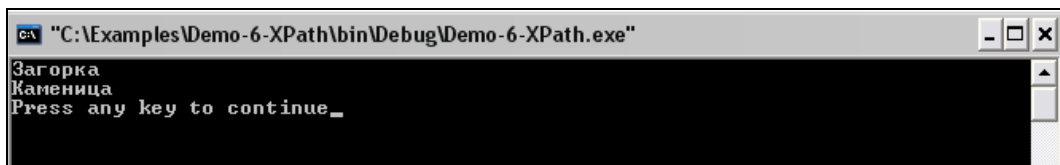
4. Извличаме всички възли, които отговарят на XPath израза и ги записваме в `XmlNodeList`:


```
XmlNodeList beerNamesList = xmlDoc.SelectNodes(xpathQuery);
```

5. Обхождаме елементите на списъка и отпечатваме имената на стоки-те, които извлякохме от XML документа:

```
foreach (XmlNode beerName in beerNamesList)
{
    Console.WriteLine(beerName.InnerText);
}
```

6. Ето съдържанието на конзолния изход след компилация и изпълнение на програмата:



Пространството System.Xml.XPath

Същинският програмен интерфейс, осигуряващ функционалност за обработка на XPath изрази, е реализиран от класа **XPathNavigator**. В действителност извикванията към методите **SelectSingleNode(...)** и **SelectNodes(...)** от класа **XmlNode** вътрешно също създават обект навигатор в процеса на своята работа.

Класът XPathNavigator

XPathNavigator е абстрактен клас, който дава възможност за навигация между отделните възли в един XML документ и изпълнение на XPath заявки върху тях.

Важно е да отбележим, че навигаторът е съвсем отделен компонент от документния клас. **XPathNavigator** работи единствено върху специална категория документни класове, известни като **XPath хранилища за данни (XPath data store)**. Тези класове представят съдържанието си под формата на XML и позволяват изпълнение на XPath заявки върху тях. Един клас в .NET Framework придобива такава възможност, имплементирайки интерфейса **IXPathNavigable**. Този интерфейс съдържа единствен метод **CreateNavigator()**, който създава инстанция на специализиран за конкретния документ навигатор (наследник на абстрактния клас **XPathNavigator**).

В .NET Framework вградените XPath хранилища са само три – **XmlDocument**, **XPathDocument** и **XmlDataDocument**.

XPathNavigator – по-важни методи

- **Select(...)** – връща множество от възли, отговарящо на зададен XPath израз. Контекстът, в който се оценява XPath заявката, е позицията на навигатора при извикването на метода.
- **SelectAncestors(...)** – връща всички предшественици на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена.
- **SelectChildren(...)** – връща всички преки наследници на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена. Атрибутите и пространствата от имена не участват в резултата.
- **SelectDescendants(...)** – връща всички наследници на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена. Атрибутите и пространствата от имена не участват в резултата.
- **MoveTo(...)** – придвижва навигатора до позицията, определена от зададения като параметър XPathNavigator обект.
- **MoveToNext(...)** – придвижва се до следващия наследник на текущия възел.
- **MoveToParent(...)** – придвижва се до родителя на текущия възел.
- **Compile(...)** – компилира XPath израз.
- **Matches(...)** – определя дали текущия възел отговаря на зададен XPath израз

XPathNodeIterator

Всеки път, когато един XPath израз поражда резултатно множество от възли (при извикване на **Select** методите), навигаторът връща като резултат нов обект от тип итератор на възли. Итераторът предоставя интерфейс за навигация в масив от възли. Базовият клас за всеки XPath итератор е класът **XPathNodeIterator**.



Функциите на един итератор лесно могат да бъдат припокрити от един XPath навигатор, но .NET Framework съзнателно предоставя функционалността им в отделни компоненти. Избраният подход на разделяне на програмните интерфейси на навигаторите и итераторите е реализиран с цел осигуряване на по-лесен достъп и обработка на XPath заявки от различни среди – XML DOM, XPath и XSLT.

XPathNodeIterator – същност

Класът `XPathNodeIterator` няма публичен конструктор и може да бъде създаван единствено от обект навигатор. Итераторът осигурява еднопосочен достъп до възлите от една XPath заявка. Итераторът не кешира информация за възлите, които обхожда – той е просто индексатор, работещ върху обект от тип навигатор, който управлява XPath заявката.

XPathNodeIterator – методи и свойства

- `MoveNext()` – придвижва се до следващия възел в множеството избрани възли на навигатора.
- `Clone()` – връща дълбоко копие на текущия `XPathNodeIterator`
- `Count` – връща броя на елементите от първо ниво (не отчита наследниците) в множеството от възли.
- `Current` – връща референция към навигатор с корен във възела, разположен на текущата позиция на итератора.
- `CurrentPosition` – връща индекса на текущия възел, избран от итератора.

XPathNavigator – пример

Ще разгледаме кратък пример, който илюстрира работата с `XPathNavigator` за обхождане и промяна на части от един XML документ. Целта на демонстрацията е да намалим цената на всички стоки от тип "бира" в XML документа с 20%.

Работният документ, който ще използваме изглежда по следния начин:

items.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Нека разгледаме необходимите действия стъпка по стъпка:

1. Стартираме VS.NET и създаваме нов проект.
2. За да получим инстанция на **XPathNavigator** обект е необходимо първо да инстанцираме нов обект от тип **XmlDocument** с работния документ **items.xml**:

```
XmlDocument xmlDoc = new XmlDocument();  
xmlDoc.Load("items.xml");
```

3. Създаваме инстанция на обект от тип **XPathNavigator**:

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

4. За да намалим цената на бирата с 20%, първо трябва да имаме лесен начин за навигация до цените на стоките от тип бира. XPath изразът, който ще използваме, търси всички възли **price**, наследници на тези възли **item**, чиито атрибут **type** има стойност **"beer"**. Възлите **item** от своя страна трябва да са преки наследници на възли **items**, закачени за коренния елемент на документа. Изпълняваме тази XPath заявка върху **XPathNavigator** обекта и получаваме **XPathNodeIterator**:

```
string xpathQuery = "/items/item[@type='beer']/price";  
XPathNodeIterator iter = nav.Select(xpathQuery);
```

5. С помощта на итератора обхождаме възлите, върнати като резултат от XPath заявката върху навигатора. Поради стремежа за разделяне на функционалността на итераторите и навигаторите, процесът на взимане стойността за всеки възел изглежда малко неестествен в началото. От итератора получаваме **XPathNavigator** обект с корен текущия възел. За да получим **XmlNode** инстанция от навигатора, извикваме метода **GetNode()** от интерфейса **IHasXmlNode** (имплементиран от **XpathNavigator**):

```
while (iter.MoveNext())  
{  
    XPathNavigator currentNode = iter.Current;  
    XmlNode xmlNode = ((IHasXmlNode) currentNode).GetNode();  
    ...  
}
```

6. Разглежданият пример чете и променя стойности на десетични числа. В XML документа те са форматираны с десетична точка, но винаги има вероятност текущата активна култура на компютъра, където изпълняваме програмата, да е различна и да форматира числата с десетична запетая (например българската култура). За да се подсигурим, че парсването на числата ще протече безпроблемно и няма да предизвика изключение от тип **FormatException**, добре е

да използваме инвариантната (културно-необвързана) култура, достъпна чрез свойството `CultureInfo.InvariantCulture`.

7. След като вече имаме достъп до цената под формата на `XmlNode`, взимаме стойността като низ и я парсваме като десетично число, с помощта на културата `CultureInfo.InvariantCulture`. Тази стойност намаляваме с 20% и я записваме като нова стойност на текущия възел:

```
public const decimal DISCOUNT = (decimal) 0.20;
...

while (iter.MoveNext())
{
    ...
    string priceStr = xmlNode.InnerText;
    decimal price = Decimal.Parse(priceStr,
        CultureInfo.InvariantCulture);
    price = price * (1 - DISCOUNT);
    xmlNode.InnerText = price.ToString(
        CultureInfo.InvariantCulture);
}
```

8. Единственото, което остава, е да запазим новия XML документ:

```
xmlDoc.Save("itemsNew.xml");
```

9. Ето как изглежда пълният сорс код на примера:

```
using System;
using System.Xml;
using System.Xml.XPath;
using System.Globalization;

class XPathNavigatorDemo
{
    public const decimal DISCOUNT = (decimal) 0.20;

    static void Main()
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.Load("../..../xml-files/items.xml");

        CultureInfo numberFormat = new CultureInfo("en-US");

        XPathNavigator nav = xmlDoc.CreateNavigator();
        string xpathQuery = "/items/item[@type='beer']/price";
        XPathNodeIterator iter = nav.Select(xpathQuery);

        while (iter.MoveNext())
```

```
{
    XPathNavigator currentNode = iter.Current;
    XmlNode xmlNode = ((IHasXmlNode) currentNode).GetNode();
    string priceStr = xmlNode.InnerText;
    decimal price = Decimal.Parse(priceStr, numberFormat);
    price = price * (1 - DISCOUNT);
    xmlNode.InnerText = price.ToString(numberFormat);
}

xmlDoc.Save("itemsNew.xml");
}
```

10. Можем да се уверим, че променена е само цената на бирата, като разгледаме документа `itemsNew.xml` след компилация и изпълнение на програмата:



```
itemsNew.xml
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.432</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.448</price>
  </item>
</items>
```

Класът XPathDocument

Класът `XPathDocument` осигурява силно оптимизиран кеш на XML документи в паметта за еднопосочна работа с XPath и XSLT в режим само за четене. Този клас е проектиран специално с цел да служи за контейнер на XPath данни и не осигурява никаква информация за възлите, които съдържа. Класът `XPathDocument` създава поддържаща мрежа от референции към възли, която позволява на XPath навигатора да работи бързо и ефективно. `XPathDocument` не зачита XML DOM спецификациите и съдържа единствен метод `CreateNavigator()` (от интерфейса `IXPathNavigable`). Удобен е при обработката на големи XML документи.

Работа с XPathDocument – пример

Търсенето на имената на всички стоки от тип бира, което демонстрирахме по-горе със средствата на XPath и `XmlNode`, лесно може да се осъществи с помощта на оптимизирания клас `XPathDocument`. Тъй като работата на класа `XPathNavigator` е напълно независима от документния клас, стъпките отново са същите, като в току-що разгледаната демонстрация:

1. Зарежда се документен клас (XPath хранилище за данни) – в този случай `XPathDocument`.
2. С негова помощ се създава навигатор.
3. Изпълнява се XPath заявка върху този навигатор (в този случай компилирана).
4. С помощта на итератор, получен от изпълнението на заявката, се обхожда множеството от възли и се изписва стойността им.

```
XPathDocument doc = new XPathDocument("items.xml");

XPathNavigator nav = doc.CreateNavigator();

XPathExpression expr = nav.Compile(
    "/items/item[@type='beer']/name");
XPathNodeIterator iter = nav.Select(expr);

while (iter.MoveNext())
{
    XPathNavigator currentNode = iter.Current;
    Console.WriteLine(currentNode.Value);
}
```

Класът XPathExpression

Класът `XPathExpression` представлява компилиран XPath израз – енкапсулация на XPath описание на път и контекст, в който ще се оценява то. `XPathExpression` няма публичен конструктор и не може да бъде създаван директно.

Компилиране на XPath изрази

XML DOM методите `SelectSingleNode(...)` и `SelectNodes(...)`, както и `Select` методите на класа `XPathNavigator` позволяват задаването на XPath заявката под формата на чист текст. XPath изразите, обаче, се изпълняват само в компилирана форма, затова тези методи прозрачно извършват компилация на XPath изразите преди да ги обработят. Потребителите могат сами да създават компилиран XPath израз с метода `XPathNavigator.Compile(...)`. Използването на компилиран XPath израз има някои предимства:

- Преизползваемост (при многократна работа с един и същи XPath израз компилация се извършва само веднъж).
- Компилацията на XPath израз позволява предварително да се знае типа на върнатата стойност (изброеният тип `XPathResultType`).

Компилираните XPath изрази могат да се използват като параметри за някои от методите на класа `XPathNavigator`, между които `Select(...)`, `Evaluate(...)`, `Matches(...)`.



XML DOM методите `SelectSingleNode(...)` и `SelectNodes(...)` не могат да приемат компилиран XPath израз като параметър. (Вътрешно те създават инстанция на `XPathNavigator`, която отново компилира текстовия XPath израз, но в този случай нямаме преизползваемост).

XSLT

След като разгледахме XPath технологията, нека се запознаем и с XSL трансформациите на XML документи.

Технологията XSLT

XSLT (Extensible Stylesheet Language Transformations) е език, който позволява трансформиране на XML документи в XML или друг текстов формат в зависимост от зададени правила.

XSLT е подмножество на езика XSL (Extensible Stylesheet Language – език за описание представянето на XML-форматирани документи), като в началото се използва за трансформации на XML елементи в комплексни стилове (например вложени таблици и индекси).

Какво представлява XSLT?

Една **XSL трансформация** е процес, при който даден XML документ се преобразува в друг текстов документ. За целта се използва XSLT шаблон, по който се извършва трансформацията.

XSL шаблон за трансформация – същност

Един **XSLT шаблон** представлява на практика поредица от шаблонни елементи. Всеки шаблон приема като вход един или повече елементи от входния XML документ и връща текстов изход въз основа на свои лите-ралаи и трансформация на приетите входни параметри.

XSLT процесорът обработва документите последователно, но разчита на XPath извиквания за извличането на възли с определени характеристики. Резултатът от една трансформация може да бъде XML документ (в частност и XHTML), HTML страница или документ във всеки един текстово-базиран формат, който отговаря на правилата, описани от трансформацията.

XSL шаблон за трансформация – пример

Нека разгледаме как в действителност изглежда един XSL шаблон за трансформация:

library-xml2html.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
  <h1>Моята библиотека</h1>
  <table bgcolor="#E0E0E0" cellspacing="1">
    <tr bgcolor="#EEEEEE">
      <td><b>Заглавие</b></td>
      <td><b>Автор</b></td>
    </tr>
    <xsl:for-each select="/library/book">
      <tr bgcolor="white">
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="author"/></td>
      </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Засага няма да навлизаме в техническите подробности и само ще споменем, че можем да използваме този шаблон за трансформация на информацията за малка домашна библиотека от XML в HTML формат.

По-важни XSLT конструкции

Конструкцията на езика XSLT се състоят от специални тагове. Те представят отделните операции, които могат да се извършват върху тагир от входния документ или върху подадените на трансформацията параметри:

- **<xsl:template match="XPath-израз">...</xsl:template>** – замества, зададената с XPath израз, част от документа с тялото на конструкцията.
- **<xsl:value-of select="XPath-израз" />** – извлича стойността на зададения XPath израз (само първото намерено съответствие).
- **<xsl:for-each select="XPath-израз">...</xsl:for-each>** – замества всеки възел, отговарящ на дадения XPath израз с тялото на конструкцията.

- `<xsl:if test="XPath-израз">...</xsl:if>` – прилага тялото на конструкцията само, ако XPath изразът се оцени с положителна булева стойност.
- `<xsl:sort select="XPath-израз" />` – В `xsl:for-each` конструкции сортира по стойността на даден XPath израз.

XSLT и .NET Framework

.NET Framework осигурява пълна XSLT поддръжка чрез класовете от пространството `System.Xml.Xsl`. Основен клас при работата с XSLT е `XslTransform`, който представлява имплементация на XSLT процесор за .NET Framework. Работата с този клас протича винаги в две стъпки – XSLT шаблонът първо се зарежда в процесора и едва тогава се извършват трансформации с него.

Класът `XslTransform` поддържа само версия 1.0 на XSLT спецификацията. Един шаблон декларира съвместимост с тази версия, като включва следното пространство от имена:

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

Важно е да се отбележи, че атрибутът `version` е задължителен, за да се осигури коректността на XSLT документа.

XslTransform – методи

Методите от класа `XslTransform`, които най-често се използват, са следните:

- `Load(...)` – зарежда XSL шаблон за трансформацията
- `Transform(...)` – извършва трансформация на даден XML документ. Приема като вход име на XML файл, `XPathNavigator` или `IXPathNavigable`. Записва изхода в XML файл, поток или `XmlWriter`.

XSL трансформация – пример

Следният кратък пример илюстрира работата с `XslTransform` класа за преобразуване на XML документ по даден XSLT шаблон. Целта, която си поставяме е да трансформираме съдържанието на XML документа `library.xml` в HTML формат.

Ето как изглежда документа `library.xml`:

`library.xml`

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
```

```

<title>Programming Microsoft .NET</title>
<author>Jeff Prosise</author>
<isbn>0-7356-1376-1</isbn>
</book>
<book>
  <title>Microsoft .NET for Programmers</title>
  <author>Fergal Grimes</author>
  <isbn>1-930110-19-7</isbn>
</book>
</library>

```

Ето и съдържанието на XSL шаблона `library-xml2html.xsl`, който описва правилата за трансформацията:

library-xml2html.xsl

```

<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
  <h1>Моята библиотека</h1>
  <table bgcolor="#E0E0E0" cellspacing="1">
    <tr bgcolor="#EEEEEE">
      <td><b>Заглавие</b></td>
      <td><b>Автор</b></td>
    </tr>
    <xsl:for-each select="/library/book">
      <tr bgcolor="white">
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="author"/></td>
      </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>

```

Шаблонът определя, че коренният елемент на входния документ (на практика това адресира целия документ, защото всички други възли са наследници на коренния елемент) трябва да се замести с тялото на HTML конструкцията, дефинирана от шаблона.

Като оставим на страна стандартните HTML елементи, интерес за нас представляват XSLT таговете `xsl:for-each` и `xsl:value-of`. Тагът `<xsl:for-each select="/library/book">` замества всеки възел от входния документ, който отговаря на зададения XPath израз с ред от таблица,

чиито колони се инициализират със стойности, извлечени от наследниците на възлите `/library/book` (`xsl:value-of` конструкциите).

Нека сега разгледаме необходимите стъпки за реализиране на програмата, която извършва XSL трансформацията:

1. Създаваме нов обект от тип `XsltTransform`:

```
XsltTransform xslt = new XsltTransform();
```

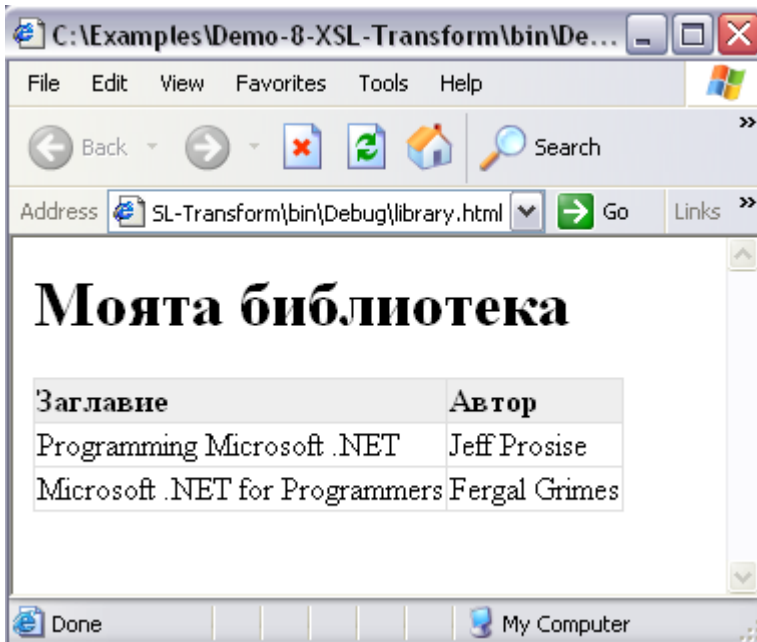
2. Зареждаме XSLT шаблона, описващ трансформацията:

```
xslt.Load("library-xml2html.xsl");
```

3. Извършваме трансформацията, като запазваме резултата във файла `library.html`. Тъй като не използваме външни XML ресурси, инициализираме третия параметър `XmlResolver` с `null`:

```
xslt.Transform("library.xml", "library.html", null);
```

4. Ето как изглежда резултатният файл `library.html` след компилация и изпълнение на програмата:



Трансформация на XML в чист текст – пример

Както знаем, XSL трансформациите могат да преобразуват даден XML документ не само в друг XML документ, но и в произволен текстов формат. За да илюстрираме това, да си поставим следната задача: Даден е XML документът:

example.xml

```
<?xml version="1.0" encoding="utf-8"?>
<values>
  <value>1</value>
  <value>2</value>
  <value>3</value>
</values>
```

Да се напише XSL шаблон, който трансформира този документ в следния текстов вид:

example.txt

```
1<2<3
```

По подразбиране XSL трансформациите преобразуват XML документ в друг XML документ. За да преобразуваме XML документ в текст, трябва да укажем в XSL шаблона следната опция:

```
<xsl:output method="text" />
```

Тя указва на XSL трансформатора да генерира изхода като чист текст вместо като XML. Сега вече за да решим поставената задача, можем да използваме следния XSL шаблон:

xml2text.xsl

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" />
<xsl:template match="/">
  <xsl:for-each select="//values">
    <xsl:for-each select="/*">
      <xsl:if test="position()>1">
        <xsl:text>&lt;</xsl:text>
      </xsl:if>
      <xsl:value-of select="." />
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

С това завършваме прегледа на средствата за работа с XML в .NET Framework. Препоръчваме Ви да не пропускайте секцията с практически упражнения, която ще Ви помогне да проверите и затвърдите познанията си относно имплементацията на XML технологиите в .NET.

Упражнения

1. Какво представлява езикът XML? За какво служи? Кога се използва?
2. Създайте XML документ `students.xml`, който съдържа структурирано описание на студенти. За всеки студент трябва да има информация за имената му, пол, рождена дата, адрес, телефон, email, курс, специалност, факултетен номер, ВУЗ, факултет, положени изпити (име на изпит, преподавател, оценка), невзети изпити, среден успех, дата на приемане във ВУЗ и очаквана дата на завършване (година и месец).
3. Какво представляват пространствата от имена в XML документите? За какво служат? Кога се използват?
4. Променете файла `students.xml` и му добавете пространство от имена по подразбиране `"urn:students"`.
5. Какво представляват XML схемите? По какво си приличат и по какво се различават DTD, XSD и XDR схемите?
6. С помощта на VS.NET създайте подходяща XSD схема за валидация на документа `students.xml`. Редактирайте генерираната схема, като внимателно съобразите всяко поле от описанието на един студент от какъв тип трябва да бъде, дали трябва да е задължително или по избор и какви са ограниченията над валидните му стойности.
7. Чрез редактора на VS.NET дефинирайте XSD схема за описание на музикален каталог. Каталогът трябва да съдържа съвкупност от албуми на различни изпълнители. За всеки албум трябва да са дефинирани: наименование, автор, година на издаване, продуцентска къща, цена и списък на песните. Всяка песен трябва да се описва със заглавие и продължителност.
8. Създайте примерен XML файл `catalog.xml`, отговарящ на описаната XSD схема. Свържете файла `catalog.xml` със съответната му схема и го валидирайте по нея с помощта на VS.NET.
9. Напишете програма, която с помощта на DOM парсера и класовете `XmlDocument` и `XmlNode` извлича от `students.xml` имената на всички студенти, които имат поне 2 невзети изпита.
10. Напишете програма, която с помощта на DOM парсера и чрез използване на хеш-таблица намира и извлича всички различни автори на музика, които се срещат във файла `catalog.xml`. За всеки автор трябва да се отпечата броя на албумите му в каталога.
11. Напишете програма, която с помощта на DOM парсера добавя даден изпит в списъка с невзетите изпити за всеки студент от файла `students.xml`. Изпитът е с фиксирано заглавие "Нов изпит" с преподавател "Нов преподавател" и трябва да се добавя само ако не се среща в списъка от взетите и в списъка от невзетите изпити за студента.

12. Напишете програма, която с помощта на DOM парсера изтрива от файла `catalog.xml` всички албуми, които струват повече от 20 лв.
13. В текстов файл в някакъв предварително известен формат са записани трите имена, адреса и телефона на даден човек. Напишете програма, която с помощта на DOM парсера създава нов XML документ, който съдържа тези данни в структуриран вид.
14. Напишете програма, която с помощта на парсера `XmlReader` извлича всички заглавия на албуми от файла `catalog.xml`.
15. Напишете програма, която с помощта на парсера `XmlReader` извлича и отпечатва за всеки студент от файла `students.xml` списък от имената на всички преподаватели, при които студентът е взел успешно някакъв изпит.
16. В текстов файл в някакъв предварително известен формат са записани трите имена, адресът и телефонът на даден човек. Напишете програма, която с помощта на класа `XmlWriter` създава нов XML документ, който съдържа тези данни в структуриран вид.
17. Напишете програма, която с помощта на класовете `XmlReader` и `XmlWriter` прочита файла `catalog.xml` и създава файла `album.xml`, в който записва по подходящ начин имената на всички албуми и техните автори.
18. Напишете програма, която претърсва зададена директория от твърдия диск и записва в XML файл нейното съдържание заедно с всичките ѝ поддиректориите. Използвайте таговете `<file>` и `<dir>` с подходящи атрибути. За генерирането на XML документа използвайте класа `XmlWriter`.
19. Напишете програма, която валидира файла `students.xml` по съответната му XSD схема.
20. Напишете програма, която с помощта на DOM модела и подходящи XPath заявки за всеки студент от документа `students.xml` извлича всичките му оценки и средния му успех и проверява дали успехът е правилно изчислен.
21. Напишете програма, която с помощта на класа `XPathNavigator` и подходящи XPath заявки извлича от файла `catalog.xml` цените на всички албуми, издадени преди 5 или повече години.
22. Напишете програма, която с помощта на XPath заявки върху DOM дървото на документа `students.xml` намира за всеки студент всички изпити, които той е взел с оценка среден (3) и променя оценката му на отличен (6).
23. Създайте подходящ XSL шаблон, който преобразува файла `catalog.xml` в XHTML документ, подходящ за разглеждане от

стандартен уеб браузър. Напишете програма, която прилага шаблона с помощта на класа **XsltTransform**.

24. Създайте XSL шаблон, който приема като вход документа **students.xml** и генерира като резултат друг XML документ, съдържащ само имената и факултетните номера на всички студенти. Напишете програма, която прилага шаблона с помощта на класа **XsltTransform**.

Използвана литература

1. Светлин Наков, Работа с XML – <http://www.nakov.com/dotnet/lectures/Lecture-12-Working-with-XML-v1.0.ppt>
2. Стоян Йорданов, Работа с XML – <http://www.nakov.com/dotnet/2003/lectures/Working-with-XML.doc>
3. MSDN Training, Introduction to XML and the Microsoft® .NET Platform (MOC 2500A)
4. XML in 10 points – <http://www.w3.org/XML/1999/XML-in-10-points>
5. MSDN Library, XML Fundamentals: Understanding XML – <http://msdn.microsoft.com/library/en-us/dnxml/html/UnderstXML.asp>
6. XML Fundamentals: Understanding XML Namespaces – http://msdn.microsoft.com/XML/Understanding/Fundamentals/default.aspx?pull=/library/en-us/dnxml/html/xml_namespaces.asp
7. XML Fundamentals: Understanding XML Schema – <http://msdn.microsoft.com/XML/Understanding/Fundamentals/default.aspx?pull=/library/en-us/dnxml/html/understandxsd.asp>
8. William J. Pardi, "XML in Action", 1999, Microsoft Press, ISBN 0735605629
9. Erik T. Ray, "Learning XML, 2nd Edition", 2003, O'Reilly, ISBN 0596004206
10. Dino Esposito, "Applied XML Programming for Microsoft .NET", 2003, Microsoft Press, ISBN 0735618011
11. Niel M. Bornstein, ".NET and XML", 2003, O'Reilly, ISBN 0596003978