Кроссплатформенные программные системы

Конспект лекций

Составитель Тимофеев А.А.

Лекция 1.

ВВЕДЕНИЕ В XML

XML – технология, предназначенная для представления текстовых данных в высокоструктурированном виде, пригодном для программной обработки. XML-документы представляют собой систему элементов, каждый элемент имеет имя и оформляется парой из открывающего и закрывающего тэгов:

```
<имя элемента>
        …              - тело элемента
</имя элемента>
```

Тело элемента может включать текст и вложенные элементы. Элемент может содержать атрибуты, которые оформляются в открывающем тэге в виде пар имя_атрибута = ‘значения_атрибута’ (в одинарных или двойных кавычках).

Пример элемента с атрибутами:

```
<имя_элемента
  атрибут 1 = “значение 1”
  атрибут 2 = “значение 2”
>
  тело
</имя_элемента>
```

Имя_элемента может быть на любом языке. Считается недопустимым отсутствие закрывающего и открывающего тэгов, кавычек у значений атрибутов, наложение элементов.

Пример недопустимого описания:

```
<элемент 1>
  <элемент 2>
</элемент1>
  <элемент 2>
```

Элементы с пустым телом допускается оформлять в виде <элемент/> это эквивалентно <элемент></элемент>

У документа должен быть только один корневой элемент, все остальные должны быть расположены так или иначе в рамках его тела. Таким образом, любой XML-документ – дерево элементов.

Кроме элементов спецификация XML определяет еще несколько специфичных понятий, которые могут использоваться при формировании XML-документа. Это:

1. Ссылки на сущности (entity references)

2. Ссылки на символы (character references)

3. Комментарии (comments)

4. Секции CDATA (CDATA sections)

5. Объявления типа документа (document type declaration)

6. Инструкция обработки (processing instruction)

7. XML-объявления (XML declaration)

8. Текст объявления (text declaration),

которые совместно с открытыми тэгами (start tags), закрывающими тэгами (end tags) формируют разметку (mark up), которая позволяет представить текст в высокоструктурированном виде. В формировании разметки также участвуют тэги пустых элементов (empty-elements tags).

Документ XML составлен из текстовых данных и разметки. Спецификация указывает ряд требований к разметке, не фиксируя при этом четко ни набор элементов, ни их атрибуты, ни правила вложенности элементов. Это сделано для того, чтобы можно было эти ограничения описывать с помощью специальных средств, в зависимости от того, данные какой предметной области должны будут представляться в виде XML в рамках приложения. В силу этого XML можно считать не столько языком, сколько технологией представления текста в структурированном виде. Поскольку XML не обладает семантикой, ссылки на сущности оформляются.

XML-документы, удовлетворяющие общим требованиям спецификации XML, называются well-formed. Если документы удовлетворяют еще и заданным спецификацией средствами ограничений, определяемыми предметной областью, то он называется valid.

Рассмотрим различные виды разметки:

Comments. <!--текст комментария -- > Предназначен для внедрения в XML-документ информации, которая не считается частью данных, переданных в составе XML-документа. XML-анализатор может включать, а может и не включать комментарии, в результате анализа – это зависит от анализатора. В тексте комментария нельзя использовать --.

Processing instructions. <? Имя_инструкции данные ?> Имя_инструкции должно отличаться от имени XML в любом сочетании регистра символов. Данные – необязательно, но если есть – нельзя использовать ?>.

Инструкции обработки. Внедряются в XML-документ для того, чтобы сообщить прикладной программе дополнительные сведения о том, как обрабатывать XML-документ. XML-анализатор обязан включать все инструкции обработки в результате анализа.

CDATA.
<![CDATA[

текст
]]>

При этом текст не должен содержать]]>. Секции CDATA используются для оформления фрагментов текста, которые содержат множество символов запрещенных спецификацией для употребления в литеральной форме в составе текста. Внутри секции CDATA они могут быть в литеральной форме.

Esc-последовательности: & - & < - < > - > “ - " ‘ - &apos. Спецификация XML запрещает использовать символы & и < в их литеральной форме везде, кроме разделителей разметки (ссылки на сущности и символы, а также тэги) или внутри комментариев, инструкций обработки и секции CDATA. Во всех остальных случаях эти символы должны заменяться на встроенные esc-последовательности. Например, в теле элемента или в значении атрибута.

ССЫЛКИ НА СУЩНОСТИ И СИМВОЛЫ

Ссылки на сущности и символы используются для замены фрагментов текста или одного символа соответственно esc-последовательности. Ссылки на символы оформляются: &#десятичный код; либо &#хшестнадцатеричный код;. Указанный десятичный или шестнадцатеричный код ссылается на UNICODE-символ.

Ссылки на сущности оформляются по-разному, в зависимости от того, где они используются. Если ссылка на сущность используется внутри XML-документа, но за пределами DTD, то: & имя сущности;. Если используется в составе DTD-описания, то: % имя сущности;. В любом случае имя сущности должно быть определено в составе DTD-описания и оно не должно быть рекурсивным. В результате анализа XML-анализатор заменяет вхождения ссылки на соответствующий символ или текст.

XML ОБЪЯВЛЕНИЯ, ТЕКСТОВЫЕ ОБЪЯВЛЕНИЯ

XML-объявления, текстовые объявления указывают, что документ является XML-документом, а не каким-то другим. При этом XML-объявление не обязательно. Текст объявления указывается в тех XML-документах, которые включаются по ссылке в другие XML-документы. Например, XML-объявление указывается в тех документах, которые не включены в другие XML-документы.

XML-объявления и текстовые объявления в общем случае выглядят одинаково:

```
<? xml
  version = “номер версии”
  encoding = “имя_кодировки_документа”
?>
```

Спецификация 1.0 – version = “1.0”

Здесь, в XML-объявлении предложение encoding – необязательно, а version – обязательно. В текстовом объявлении оба предложения - необязательны. Если не указана кодировка, используется utf-8 (UNICODE). Если не указывается версия, то используется 1.0.

DTD (DOCUMENT TYPE DECLARATION)

Ограничения на структуру XML-документов, специфичные для той или иной предметной области, могут быть описаны формально одним из двух способов:

1. предполагает использование языка DTD (document type declaration)

2. использование XMLSchema

Язык DTD определен в составе спецификации XML, является простым языком с довольно ограниченными возможностями. XMLSchema описывается на XML-языках, которые определяются отдельными спецификациями. Комитетом W3C в качестве международного стандарта был утвержден язык описания XMLSchema.

DTD-описания могут включаться непосредственно в состав XML-документа, а могут оформляться и отдельно от него. В последнем случае в XML-документ может быть внедрена ссылка на DTD-описание.

Допускается также одновременно внедрить ссылку на DTD и само DTD-описание. В этом случае непосредственное присутствие DTD-описания имеет приоритет над ссылкой.

Для внедрения DTD-описания непосредственно или по ссылке используется document type declaration. Оформление:

1. по ссылке (external subset)
<! DOCTYPE имя_корневого_элемента
ссылка
>

2. internal subset
<! DOCTYPE имя_корневого_элемента [
DTD-описание
] >

3. совмещение, причем internal имеет приоритет над external
<! DOCTYPE имя_корневого_элемента ссылка [
DTD-описание
] >

ОБОБЩЕННАЯ СТРУКТУРА XML-ДОКУМЕНТА

В общем, документ можно представить из последовательности 3-х частей: пролог, корневой элемент, разное. Пролог включает последовательно необязательное XML-объявление, комментарии и инструменты обработки. Среди них может присутствовать одно объявление типа документ. Корневой элемент содержит структурные данные и может содержать текст, вложенные элементы, комментарии, секции CDATA, инструменты обработки. После корневого элемента могут идти комментарии и инструменты обработки.

Любые элементы разметки могут оформляться пробельным материалом white space: пробел (%#x20;), табуляция (&#x9), перевод каретки (&#xD), возврат строки (&#xA).

Расположение и интерпретация ссылок на символы и сущности регламентируется подробным набором правил (см далее отдельно).

XML-документы являются регистрозависимыми. Кроме того, весь пробельный материал, не являющийся частью разметки, сохраняется.

Обработка символов конца строки ведется следующим образом: любая последовательная пара символов

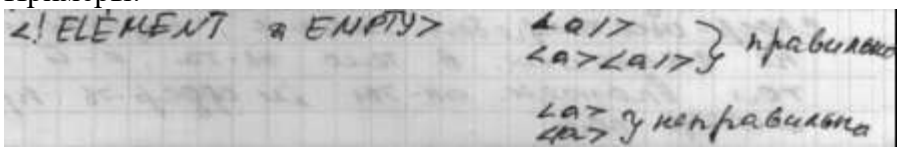
 а также
 без &#xA преобразуется в &#xA.

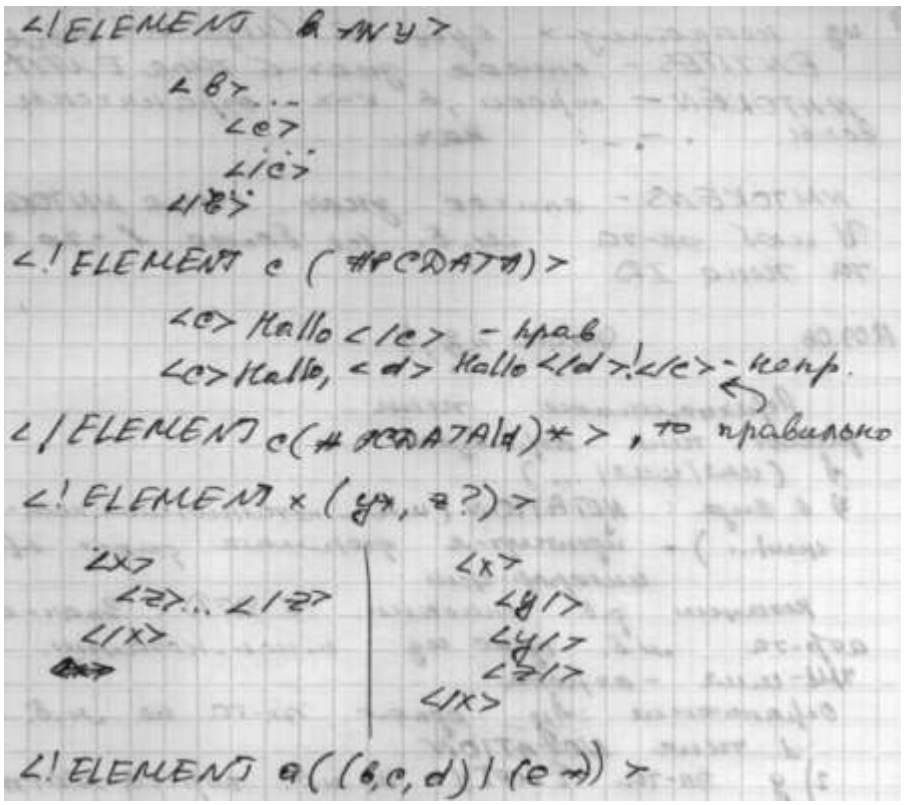
ЯЗЫК DTD

Язык DTD представляет собой последовательность объявлений и ссылок на параметризованные сущности (parameter entity reference). Объявления могут содержать: объявления элементов, атрибутов, сущностей, нотаций, а также инструкции обработки и комментарии.

ОПИСАНИЕ ЭЛЕМЕНТОВ

<!ELEMENT имя_элемента спецификация_тела> Спецификация тела может указывать, что: элемент пустой (EMPTY); ограничение на тело отсутствует (ANY) либо задавать некую модель тела. Модель задается с использованием конструкций последовательности и выбора, с которыми можно использовать модификаторы повторения: ? – необязательное присутствие элемента; + - элемент присутствует один или более раз последовательно; * - элемент присутствует 0 или более раз последовательно. Последовательность задается с помощью конструкции: (имя_элемента, имя_элемента, …). Выбор: (имя_элемента/имя_элемента/…). Конструкции повтора и выбора можно комбинировать произвольным образом. Чтобы указать, что в теле элемента может быть только текст, указывается #PCDATA. Чтобы указать, что в теле элемента могут произвольным образом смешиваться текст и элементы, используют: (#PCDATA/имя_элемента/имя_элемента/…)*
Примеры:





#ОПИСАНИЕ АТРИБУТОВ

Список атрибутов для элемента можно описать так: <!ATTLIST имя_элемента список_определений>. Список определений содержит разделенные пробелами конструкции вида: имя_атрибута тип_атрибута умолчание, каждая из которых описывает один атрибут элемента с указанным именем. Допускается каждый атрибут описывать индивидуально своим ATTLIST. У них имя атрибута будет одним и тем же. Тип атрибута может быть встроенным, либо перечислимым.

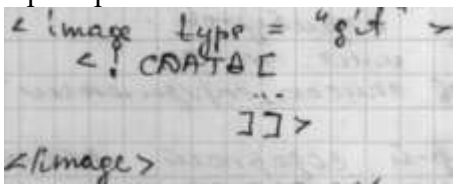
Встроенные типы: CDATA – значением атрибута является произвольный текст; ID – указывает, что значением атрибута является некоторое имя, которое в пределах документа уникально (среди любых элементов с атрибутами этого типа); IDREF – значением атрибута является значение атрибута типа ID какого-то другого элемента; IDREFS – представляет список значений типа IDREF, элементы списка разделяются пробелами; ENTITY – значением атрибута является имя неанализируемой сущности (unparsed entity), которая описана в DTD; ENTITIES – значениями является список значений типа ENTITY, которые разделяются между собой пробелами; NMTOKEN – тип указывает, что значением атрибута является строка, состоящая из букв, цифр и символов . - _ : ; NMTOKENS значение – список значений типа NMTOKEN, разделенных пробелом. Ограничение: у любого элемента может быть атрибут типа ID, но только один!

Перечислимые типы задают список возможных значений атрибута. Задаются одним из 2-х способов:

1. (имя|имя|…);

2. NOTATION (имя_нотации|имя_нотации|…), в этом случае каждое из имен должно быть описано как нотация в DTD.

Ограничение: может быть не более одного атрибута типа NOTATION, а у элементов, объявленных как EMPTY, атрибутов типа NOTATION быть не может.
Пример:



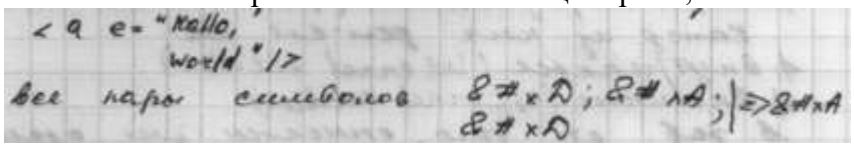
Атрибуты типа NOTATION используются для связывания с элементами некоторого публичного или системного идентификатора, который может быть использован приложением для идентификации тела элемента.

Умолчания описываются следующим образом: #REQUIRED – атрибут обязателен; #IMPLIED – значение по умолчанию не указано и атрибут необязателен; #FIXED – атрибут обязательно должен быть указан с этим значением (имя_атрибута тип_атрибута #FIXED значение). Атрибут типа ID должен быть описан как #REQUIRED либо как #IMPLIED. Заданное значение по умолчанию должно соответствовать типу атрибута. Значение по умолчанию указывается в одинарных или двойных кавычках.

ПРОЦЕДУРА НОРМАЛИЗАЦИИ ЗНАЧЕНИЙ ЭЛЕМЕНТОВ

Значения атрибутов в документе при анализе подвергаются процедуре нормализации, которая выполняется так:

1. выполняется обработка символов конца строки;

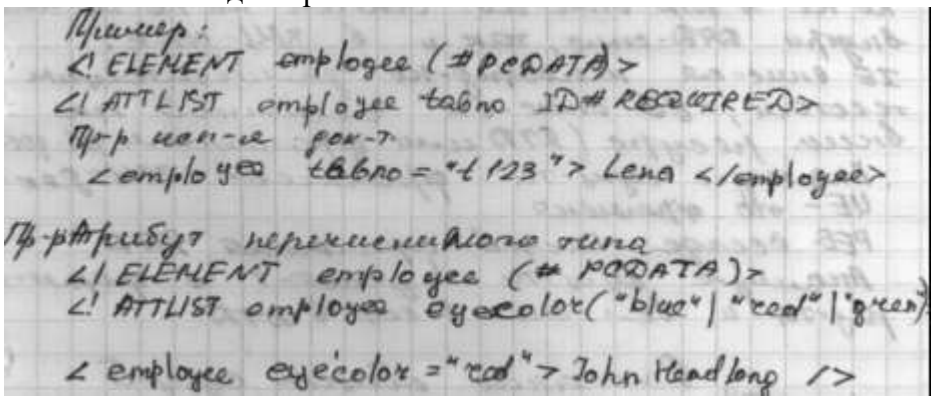


2. выполняется замена ссылок на символы собственно символами;

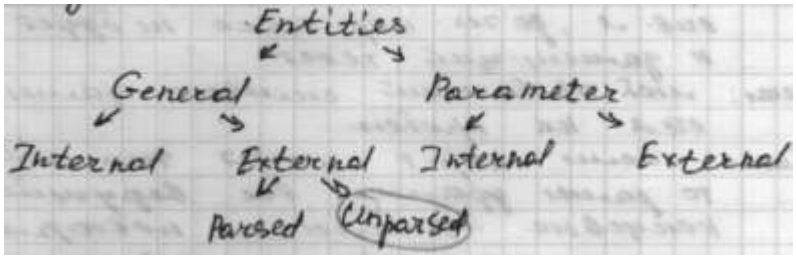
3. каждая ссылка на сущность обрабатывается рекурсивно до тех пор, пока не будет вычислен замещающий ее текст;

4. любой пробельный символ заменяется на пробел.

5. если тип атрибута не CDATA, то значение атрибута далее обрабатывается так: удаляются все ведущие и концевые пробелы, а все последовательности пробелов заменяются на один пробел.



СУЩНОСТИ И ССЫЛКИ НА НИХ



Сущности бывают двух типов: общие (general) и параметризованные (parameter). Сущности каждого вида делятся на внутренние (internal) и внешние (external).

General external entities дополнительно делятся на анализируемые (parsed) и неанализируемые (unparsed). Различия между общими и параметризованными сущностями заключаются в том, что ссылка на общую сущность может присутствовать в XML-документе и в DTD, а ссылка на параметризованную сущность – внутри DTD-описания.

Для внутренних сущностей замещающий текст указывается непосредственно при описании сущности. Для внешних - указывается ссылка на ресурс, который содержит замещающий текст. Различия между parsed и unparsed:

1. parsed включается в состав XML-документа и участвует при формировании результата анализа;

2. unparsed ссылаются на внешние ресурсы произвольного содержания. Обработка таких ресурсов выполняется не XML-анализатором, а приложением.

ОПИСАНИЕ ОБЩИХ СУЩНОСТЕЙ

<!ENTITY имя_сущности спецификация_сущности> . Спецификация сущности может задаваться значением сущности в одинарных или двойных кавычках, может содержать ссылки на символы, а также общие и параметризованные сущности. Спецификация может указывать ссылку на ресурс: SYSTEM uri либо PUBLIC публичный_идентификатор uri. Uri содержит ссылку на физический ресурс (в том или ином формате (URI, URM)), который будет использован для извлечения значения сущности.

Публичный идентификатор – некий идентификатор, который принято составлять следующим образом: -//W3C//язык_описания имя_проекта// язык. Например, для HTML документа: -//W3C//DTD HTML 4.01//EN

После ссылки может идти указание нотации в виде NDATA – имя нотации. Если указание нотации присутствует, то эта сущность general external unparsed, иначе general external parsed. При этом используемое здесь имя нотации должно быть объявлено. Если значение сущности указано при объявлении, то она general external.

ОПИСАНИЕ ПАРАМЕТРИЗОВАННЫХ СУЩНОСТЕЙ

Описание параметризованных сущностей выполняется также, как и общих, но с двумя отличиями:

1. перед именем сущности ставится %;

2. при описании external сущностей нельзя использовать указание нотации, так как все параметризованные сущности считаются parsed.

Замечание: ссылка на DTD в document type declaration оформляется по тем же правилам, что и ссылка на значение внешней сущности.

Ссылки на сущности используются с определенными ограничениями:

1. имя сущности должно быть описано;

2. сущность должна быть parsed;

3. имена неанализируемых сущностей могут быть использованы только как значения атрибутов типа ENTITY и ENTITIES;

4. ссылки на параметризованные сущности не должны появляться за пределами DTD;

5. анализируемые сущности не должны содержать ссылок на самих себя;

6. внешние анализируемые сущности должны начинаться с text declaration.

При использовании ссылок на сущность нужно следить за тем, чтобы после замещения их на соответствующим текстом, результирующий XML-документ оставался well formed.

ОПИСАНИЕ НОТАЦИЙ

Объявление нотаций вводит имена, которые могут использоваться при описании общих внешних неанализируемых сущностей или тела элемента, содержащего атрибут типа NOTATION. Обработка такого рода содержимого должна выполняться приложением. <!NOTATION имя_нотации ссылка>. Ссылка может содержать системный идентификатор: SYSTEM url либо PUBLIC публичный идентификатор, либо то и другое вместе: PUBLIC публичный идентификатор URL.

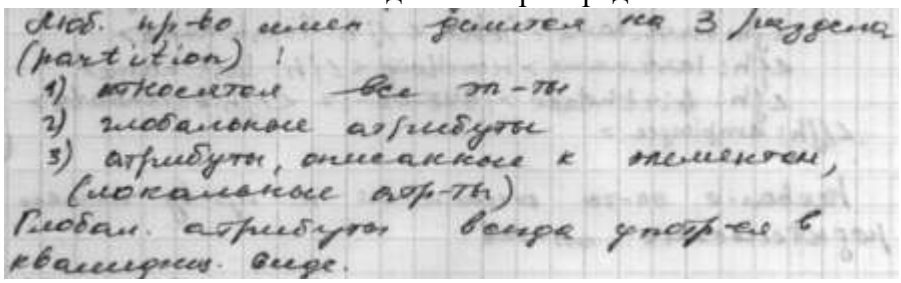
ПРОСТРАНСТВО ИМЕН XML

XML-документы могут конструироваться из элементов и атрибутов, определенных в разных схемах или DTD. Например, WSDL-описание web-сервисов, где в рамках одного документа используются конструкции, определенные схемами языков WSDL, W3C XMLSchema, SOAP. Так как разработка схем может вестись независимо, при совместном их использовании возникает опасность конфликта имен элементов и атрибутов. Для преодоления этой трудности был придуман механизм пространств имен XML (XML namespaces). Суть механизма – с каждой схемой сопоставляется некоторый глобально-уникальный идентификатор, задаваемый в формате uri. Этот uri служит именно для идентификации пространства имен, а не для извлечения фактической схемы на этапе валидации XML-документа. Это означает, что идентификатор пространства имен может и не соответствовать никакому актуальному ресурсу в сети.

Идея заключается в том, что идентификатор пространства имен совместно с именем элемента или атрибута из этого пространства имен образует глобально-уникальное имя, так как имена элементов и атрибутов в пределах одного пространства имен – уникальны. Чтобы избавить от необходимости квалификации каждого элемента и атрибута сравнительно длинным uri, вместо них используют короткий префикс, который вводится в XML-документе по определенным правилам. Выполняется это с помощью атрибутов, которые имеют имя xmlns либо имеют префикс xmlns: и локальное имя. Атрибуты с именем xmlns используются для объявления пространства имен по умолчанию, а атрибуты с префиксом xmlns: используются для объявления префиксов, занимающих в данном XML-документе соответствующий идентификатор пространства имен. Сам префикс вводится локальным именем: xmlns: локальное_имя.

Введенные префиксы используются для квалификации элементов и атрибутов в виде: <ns prefix>: <local name>.

Элементы без префиксов считаются относящимися к ns по умолчанию. Атрибуты без префиксов считаются относящимися к тому пространству имен, к которому относится элемент, с которым они используются. Префиксы пространств имен и пространства имен по умолчанию можно вводить в любом элементе. При этом действие вводимых префиксов и умолчаний распространяется на все вложенные в данный элемент элементы и атрибуты, если только в них не было сделано переопределение.


Любое пр-во имен делится на 3 раздела (partition):
1) относятся все эл-ты
2) глобальные атрибуты
3) атрибуты, относящиеся к элементам, (локальные атр-ты)
Глобал. атрибуты всегда употр. в квалифиц. виде.

Пример:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<definitions
  xmlns=”http://shemas.xmlsoap.org/wsdl”
  xmlns:soap=” http://shemas.xmlsoap.org/wsdl/soap/”
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:tns=http://johnheadlong.nightmail.ru/schemas/
>
  <types>
    <xsd:schema>
      <xsd:complexType name=”…”>
        … … …
      </xsd:comlexType>
        … … …
    </xsd:schema>
  </types>
</definitions>
```

Лекция 2.

ЯЗЫК СХЕМ

Язык схем – это XML-язык, предназначенный для описания структуры XML-документов. Сами описания, выполненные на языке схем, являются XML-документами. Элементы, атрибуты языка схем определены пространством имен http://www.w3.org/2001/XMLSchema

Существенное отличие языка схем от DTD – здесь введено понятие «тип»: элементы и атрибуты определяются тем или иным типом. Важно то, что типы могут быть описаны отдельно от элементов и атрибутов. Язык схем поддерживает богатые возможности для описания новых типов на базе встроенных либо описанных на языке схем типов.

Типы бывают простыми и сложными. Значения простого типа задаются строкой, не содержащей разметки. Значения сложного типа формируются из элементов и атрибутов. Атрибуты могут быть только простых типов, а элементы еще и сложных.

Корневым элементом в языке схем является элемент schema, внутри которого описываются типы, элементы и атрибуты.

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

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://isim.vlsu.ru/schema/order"
        xmlns:tns="http://isim.vlsu.ru/schema/order"
        elementFormDefault="qualified">

  <complexType name="orderType">
    <sequence>
      <element name="customer" type="tns:customerType"/>
      <element name="lineItems" type="tns:lineItemType"/>
    </sequence>
  </complexType>

  <element name="order" type="tns:orderType"/>
  <complexType name="customerType">
    <sequence>
      <element name="firstName" type="string"/>
      <element name="lastName" type="string"/>
    </sequence>
  </complexType>
  <complexType name="lineItemsTypeType">
    <sequence>
      <element name="lineItem" type="tns:lineItemType" maxOccurs="20"/>
    </sequence>
  </complexType>
  <complexType name="lineItemType">
    <sequence/>
     <attribute name="sku" type="positiveInteger"/>
     <attribute name="quantity" type="double"/>
   </complexType>
```

</schema>

Пример XML-документа, составленный в соответствии с этой схемой:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<po:order
  xmlns:po==”http://www.johnheadlong.nightmail.ru/schemas/”
  xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
  xsi:schemaLication=file:///home/john/order.xsd
>
  <po:customer>
    <po:firstName>John</po:firstName>
    <po:lastname>Headlong</po:lastName>
      … … …
  </po:customer>
  <po:lineItems>
    <po:lineItem sku=”123” quantity=”5”/>
    <po:lineItem sku=”456” quantity=”25”/>
  </po:lineItems>
</po:order>
```

ОПИСАНИЕ СЛОЖНЫХ ТИПОВ, ОПИСАНИЕ ЭЛЕМЕНТОВ И АТРИБУТОВ

Для описания сложных типов используется complexType. Если сложный тип описывается отдельно от элемента, то он обязательно снабжается именем, которое задается атрибутом name. Это имя поступает в пространство имен, идентификатор которого указывается атрибутом targetNamespace элемента schema. По этому имени (в его квалифицированной форме) можно ссылаться на соответствующий тип при описании элементов.

Для того чтобы имена, определяемые схемой, можно было использовать в квалифицированной форме необходимо ввести префикс (в качестве которого принято использовать префикс tns) для идентификатора целевого пространства имен, заданного атрибутом Namespaces. Этот префикс вводят в элементе schema. В результате, открывающий тэг schema выглядит:

```
<schema
  xmlns=”http://www.w3.org/2001/XMLSchema”
  targetNamespace=”целевое_пространство имен”
  xmlns:tns=”целевое_пространство_имен ”
>
```

Комплексные типы можно описать и при описании элемента, тогда имя типа не указывается. Соответствующие типы называются анонимными. Разница: именованные типы можно использовать многократно для описания любых элементов (элементов с любыми именами) одного типа.

Пример использования именованного типа:

```
<complexType name=”customerType”>
    <sequence>
      <element name=”firstNamer” type=”string”/>
      <element name=”lastName” type=”string”/>
```

```
        … … …
    </sequence>
</complexType>
<element name=”customer” type=”tns:customerType”/>
```

Пример анонимного типа:

```
<element name=”customer”>
  <complexType name=”customerType”>
    <sequence>
      <element name=”firstNamer” type=”string”/>
      <element name=”lastName” type=”string”/>
        … … …
    </sequence>
  </complexType>
</element>
```

Внутри элемента complexType описывается модель содержимого, которая является значением этого типа. Модель формируется из элементов sequence, choice и некоторых других специальных элементов.

Частью определения модели является и описание атрибутов. Элементы sequence и choice могут употребляться в произвольных сочетаниях, при этом sequence определяет последовательность элементов, указанных в его теле, а choice – выбор одного из элементов, указанных в его теле.

Сами элементы при этом описываются элементом element. Имя элемента задается атрибутом name, а тип задается либо по имени атрибутом type в квалифицированной форме, либо анонимным описанием его в теле элемента element.

```
<complexType name=”customerType”>
  <sequence>
    <element name=”publication” maxOccurs=”unbounded”>
      <complexType>
        <sequence>
          <choice>
            <element name=”isbn” type=”tns:isbnType”/>
            <element name=”bbk” type=”tns:bbkType”/>
          </choice>
          <element name=”author” type=”string”/>
            … … …
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<element name=”publications” type=”tns:publicationsType”/>
```

Пример фрагмента XML-документа, соответствующий этой схеме:

```
<publications>
  <publication>
```

```
    <isbn>532-321-028</isbn>
    <author>John Headlong</author>
  </publication>
  <publication>
    <bbk>5.371B</bbk>
    <author>Alex Smith</author>
  </publication>
    … … …
</publications>
```

При описании элементов можно использовать атрибуты minOccurs и maxOccurs, которые контролируют повторяемость элемента и его обязательность. По умолчанию эти атрибуты равны 1, что означает, что элемент обязателен и должен присутствовать в единственном экземпляре. Для описания необязательного элемента достаточно указать minOccurs=”0”. Для указания неограниченной возможности повторения: maxOccurs=”unbounded”.

Пример, когда элемент необязателен:

<element name=”comment” type=”string” minOccurs=”0”/>

Пример, когда может быть от 5 до 20 элементов заказа:

<element name=”lineItem” type=”lineItemType” minOccurs=”5” maxOccurs=20”/>

Пример, когда элемент необязателен, но их количество неограниченно:
<element name=”message” type=”string” minOccurs=”0” maxOccurs=”unbounded”/>

Для элемента можно задать значение по умолчанию атрибутом default. При этом используются правила:

1. если элемент присутствует в документе и имеет некоторое значение (значением элемента является его тело), то оно считается значением элемента.

2. если элемент существует в документе, но его тело пустое, то его значением считается значение по умолчанию, указанное при описании элемента

3. если элемент отсутствует в документе, то заданное значение по умолчанию не изменит этой ситуации, то есть элемент считается отсутствующим.

Пример: <element name=”age” type=”positiveInteger” default=”18”/>. В XML: <age/> (не является ошибкой, так как есть default в схеме) - <age>18</age>.

При описании элементов может использовать атрибут fixed для указания фиксированного значения. Смысл этого значения: соответствующий элемент должен иметь указанное значение и никакое другое. Смысл default и fixed исключает их одновременное использование при описании элементов. <element name=”country” type=”countryType” fixed=”US”/> указывает, что элемент country должен обязательно присутствовать в документе и иметь значение US: <country>US</country>. При этом элемент может быть сделан необязательным: <element name=”country” type=”countryType” minOccurs=”0” fixed=”US”/>. Default и fixed имеет смысл использовать лишь для элементов простых типов, так как указываются для атрибутов лишь простые значения.

Атрибуты описываются элементов attribute. Имя атрибута указывается атрибутом name, тип – type. Имя типа – в квалифицированной форме.

При описании атрибутов можно использовать атрибут use, который контролирует обязательность или необязательность атрибута и имеет три значения: required – обязательный, optimal – необязательный, prohibited – запрещенный. Запрещенный атрибут подобен запрещенному элементу, который описан с ограничениями minOccurs=”0”, maxOccurs=”0”.

При описании атрибутов могут использоваться default и fixed, но не вместе. Смысл default при описании атрибутов:

1. если атрибут присутствует, то его значением считается явно указанное значение (даже если оно является пустой строкой)

2. если атрибут отсутствует, то его значение – значение по умолчанию.

Элементы и атрибуты, описанные непосредственно в элементе schema, называются глобальными. При этом они могут выступать в роли корневых в XML-документах, построенных на базе schema.

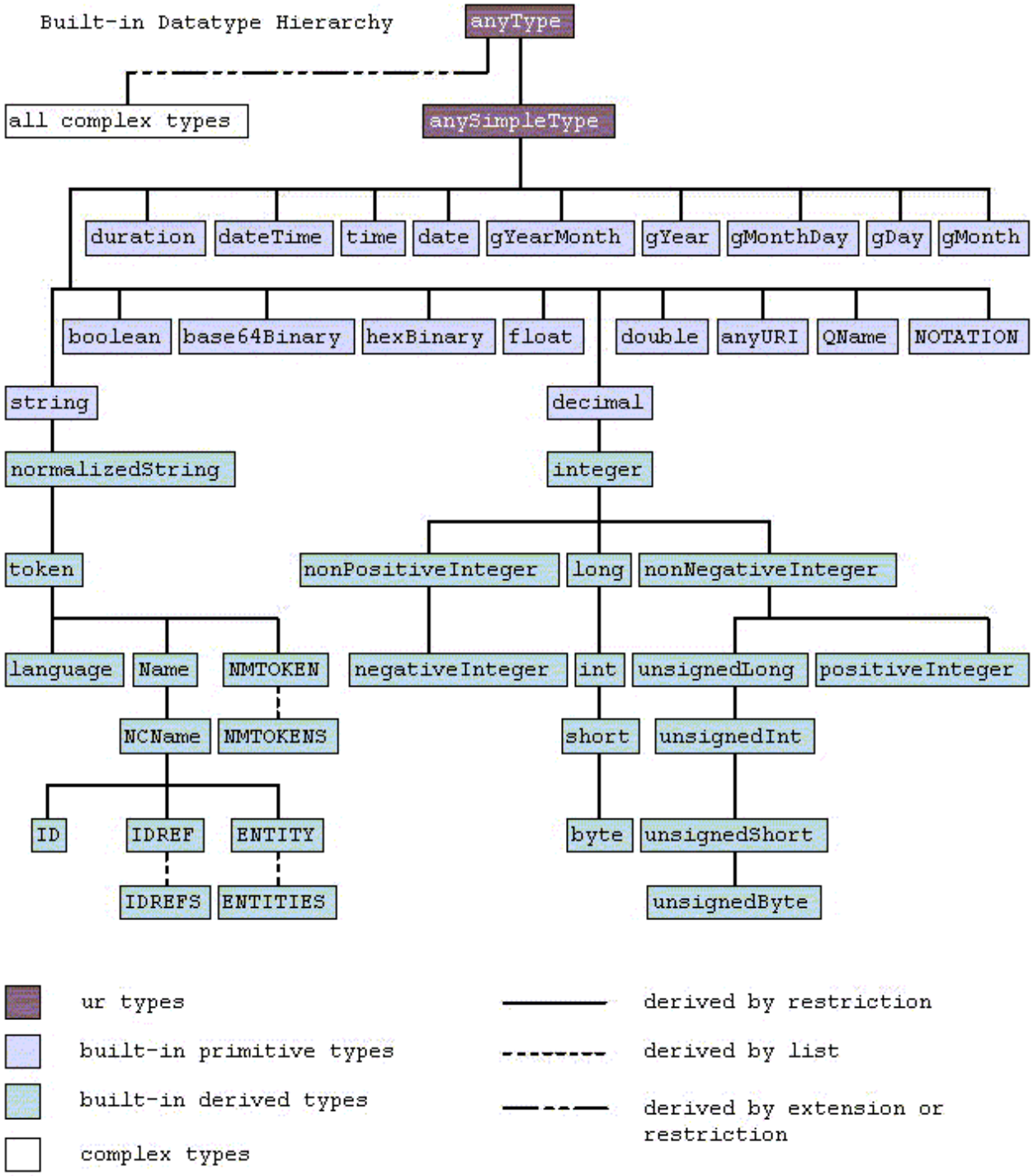
На глобальные элементы и атрибуты можно ссылаться в других фрагментах схемы с помощью атрибута ref, который не может быть использован при описании самих глобальных элементов или атрибутов. Их тип должен быть указан анонимно или явно атрибутом type. При описании глобальных элементов нельзя использовать minOccurs и maxOccurs, а для атрибутов – ограничение use.

При использовании элемента по ссылке тело должно быть пустым.

```
<element name=”comment” type=”string”/>
<complexType name=”…”>
  <sequence>
    …
    <element ref=”comment” minOccurs=”0”/>
  </sequence>
</complexType>
```

ОПИСАНИЕ ПРОСТЫХ ТИПОВ

Описание простых типов основывается всегда на ограничении одного из базовых типов (базовый тип должен быть простым). Имеется набор встроенных простых типов. Наиболее важные: string – строка, совместимая со спецификацией XML, byte, unsignedByte, short, unsignedShort, int, unsignedInt, long, unsignedLong, positiveInteger, negativeInteger, nonPositiveInteger, nonNegativeInteger, float, double, Boolean, time, date, dateTime, base64Binary – для представления бинарных данных в соответствии с кодировкой base64, hexBinary, anyURI – для представления URL, language – для представления данных о стране и языке (en – GB, en – US, ru - RU). Набор типов для совместимости со спецификацией XML: ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS. Name – имя в терминологии XML 1.0. QName – для представления квалифицированных имен с префиксов. NCName – для представления локальных имен без префикса Namespaces.



Описание производных простых типов выполняется элементом simpleType. Имя типа задается атрибутом name. Новый простой тип может быть введен путем описания ограничения базового типа (вложенным элементом restriction) либо как список значений (list) или объединение (union).

При описании ограничений базовый тип указывается атрибутом base, а сами ограничения описываются в виде фасетов (facets).

Для каждого встроенного типа, используемого как базовый, применим свой набор фасетов.

<simpleType name=”ageType”>

```
  <restriction base=”positiveInteger”>
    <minInclusive value=”18”/>
    <maxInclusive value=”45”/>
  </restriction>
</simpleType>
```

Фасеты задаются элементами, имя которых соответствует имени фасета, значение фасета – атрибутом value.

```
<simpleType name=”skuType”>
  <restriction base=”string”>
    <pattern value=”\d{3}-[A-Z]{2}”/>
  </restriction>
</simpleType>
```

Для создания перечисляемых типов:

```
<simpleType name=”usaStateType”>
  <restriction base=”string”>
    <enumeration value=”CA”/>
    <enumeration value=”NY”/>
      … … …
  </restriction>
</simpleType>
```

Фасеты и их значение: length – ограничивает длину некоторого типа (minLength, maxLength), pattern – задает шаблонб enumeration – задает перечисление, maxInclusive и minInclusive – максимум и минимум включая число, maxExclusive и minExclusive – максимум и минимум не включая, totalGigits – общее количество в числе, fractionDigits – общее количество в дробной части.

Списочные значения – строки, которые содержат элементы списка, разделенных символом пробела. Описываются элементов list, элементы задаются атрибутом itemType.

```
<simpleType name=”usaStateType”/>
  <list itemType=”usaStateType”/>
</simpleType>
```

Не списочные типы могут накладывать ограничения (restriction) в производных типах.

Объединения подобны записи с вариантами в Pascal, то есть значением объединения может быть значение одного из типов, входящих в объединение. Типы, входящие в объединение, должны быть простыми. Описывается элементов union. Типы, входящие в объединение указываются атрибутом memberTypes, значение которого – список имен типов, входящих в объединение.

```
<simpleType name=”bbk”>
  <restriction base=”string”>
    <pattern value=”…”/>
  </restriction>
</simpleType>
<simpleType name=”isbn”>
```

```
  <restriction base=”string”>
    <pattern value=”…”/>
  </restriction>
</simpleType>
```

Эквивалентно:

```
<simpleType name=”bookIDType”>
  <union memberTypes=”bbl isbn”/>
</simpleType>
<complexType name =”bookType”/>
  <sequence>
    … … …
  </sequence>
  <attribute name=”bookId” type=”bookIdType”/>
</complexType>
```

ОПИСАНИЕ СЛОЖНЫХ ТИПОВ НА БАЗЕ СУЩЕСТВУЮЩИХ

Язык схем предлагает для этого две возможности: расширение (extension) и ограничение (restriction). Причем в качестве базовых могут выступать как простые, так и сложные типы. В первом случае используется механизм расширения, который позволяет описать атрибуты (являющиеся частью определения сложного типа) для элемента со значением простого типа.

```
<simpleType name=”currencyType”>
  <restriction base=”string”>
    <pattern value=”…”/>
  </restriction>
</simpleType>
<simpleType name=”currencyCodeType”>
  <restriction base=”string”>
    <enumeration value=”USD”/>
      … … …
  </restriction>
</simpleType>
```

Описание элемента первого типа со значением второго типа:

```
<complexType name =”sumType”/>
  <simpleContent>
    <extension base=”currencyType”>
      <attribute name=”currency” type=”currenceCodeType”/>
    </extension>
  </simpleContent>
</complexType>
```

При использовании сложных типов как базовых, возможно и расширение и ограничение. Расширение заключается в том, что к базовому типу добавляются описания новых вложенных элементов и атрибутов. Делается это с помощью элемента complexType.

```
<complexType>
  <sequence>
    <element name=”login” type=”string”/>
    <element name=”password” type=”hexBinary”/>
  </sequence>
  <attribute name=”id” type=”ID”/>
</complexType>
<complexType name=”customerType”>
  <complexContent>
    <extension base=”userType”>
      <sequence>
        <element name=”firstName” type=”string”/>
        <element name=”lastName” type=”string”/>
          … … …
      </sequence>
      <attribute name=”lastlogin” type=”dateTime”/>
    </extension>
  </complexContent>
</complexType>
```

При описании сложных типов путем ограничения в произвольном типе описываются дополнительные ограничения для типов элементов, самих элементов и атрибутов. Эти ограничения должны быть более жесткими по сравнению с базовыми типами для того, чтобы значения новых типов были совместимы со значениями базовых типов. Так же как и производные типы, описанные путем расширения.

Смысл совместимости состоит в том, что код, предназначенный для работы со значениями базовых типов, должен также хорошо работать и со значениями производных типов без изменений.

Ограничения могут касаться повторяемости элементов (атрибуты maxOccurs, minOccurs), указания более узких по возможным значениям простых типов для элементов и атрибутов, задания значения по умолчанию или фиксированных значений, не указанных в базовом типе.

```
<complexType name=”listItemsType”>
  <sequence>
    <element name=”listItem” type=”listItemType” minOccurs=”0” maxOccurs=”unbounded”/>
  </sequence>
  <attribute name=”owner” type=”ownerType”/>
</complexType>
```

производный тип:

```
<complexType name=”restrictedListItemsType”>
  <complexContent>
    <restriction base=”listitemsType”>
      <sequence>
        <element name=”listItem” type=”listItemType” minOccurs=”1” maxOccurs=”10”/>
      </sequence>
      <attribute name=”owner” type=”ownerType” use=”required” fixed=”John”/>
```

```
    </restriction>
  </complexContent>
</complexType>
```

Язык схем поддерживает концепцию абстрактных типов. Она подобна той, что поддерживается в ООЯ, то есть при описании элемента его типам может быть указан абстрактный тип (тип, описан как абстрактный). При этом в XML-документе использование данного элемента предполагает, что его значение будет соответствовать значению одного из производных типов, расширяющих абстрактный, а фактический тип элемента в документе должен быть указан с помощью атрибута type, который определен в пространстве имен http://www.w3.org/2001/XMLShema-instance, которому принято назначать префикс xsi.

Указанное пространство имен содержит имена, которые предназначены для использования не в схемах, а в XML-документах.

```
<complexType name “publicationType” abstract=”true”/>
<complexType name “bookType”>
  <complexContent>
    <extension base=”publicationType”>
      … … …
    </extension>
  </complexContent>
</complexType>
<complexType name “periodicalType”>
  <complexContent>
    <extension base=”publicationType”>
      … … …
    </extension>
  </complexContent>
</complexType>
<element name=”publication” type=”publicationType”/>
```

XML-документ:

```
<root xmlns=”…”
   xmlns:xsi=”http://www.w3.org/2001/XMLShema-instance”
>
  …
<publication xsi:type=”bookType”>
  …
</publication>
<publication xsi:type=”periodicalType”>
  …
</publication>
  …
</root>
```

При описании как простых, так и сложных типов существует возможность ограничить способы описания производных типов. Эти ограничения описываются атрибутом final, который можно использовать совместно с элементами как simpleType, так и complexType.

Для запрета производных типов вообще используется значение #all. Для запрета расширения используется значение – extension. Для запрета любых описаний путем ограничений – restriction.

Кроме того, существует возможность запрещать ограничения индивидуально для каждого фасета в базовом типе: fixed=”true”. Существует возможность задать значение атрибута final для всех описаний любых типов в схеме глобально с помощью атрибута finalDefault элемента schema.

Язык схем поддерживает аналог null значений языка SQL или значений null/nil в языках программирования. Чтобы элемент мог принимать null значения при его описании необходимо указать атрибут nillable со значением true.

При этом в документе этот элемент может быть пустым – это полезно, когда значение элемента неизвестно, но факт его присутствия важен или требуется схема. В этом случае у такого элемента должен быть указан атрибут xsi:nil=”true”. Такой элемент не может иметь тела, но может иметь атрибуты.

При описании сложных типов, значения которых содержит как элементы, так и текст (mixed content) используется атрибут mixed со значением true.

```
<comlexType name=”paraType mixed=”true”>
  <sequence>
    <element name=”em” type=”string” minOccurs=”0” maxOccurs=unbounded”/>
  </sequence>
  <element name=”para” type=”paraType”/>
</complexType>
```

instance:
<para>Hallo,Dr.John! I’m verygladto see you!</para>

Смысл такого описания mixedContent отличается от соответствующего описания в DTD, так как в DTD нет возможности ограничить ни порядок следования, ни повторяемость вложенных элементов, окруженных текстом.

В сложных типах с атрибутом mixed все эти ограничения имеют значение:

```
<complexType name=”paraType”>
  <sequence>
    <element name=”em”/>
    <element name=”strong” type=”string”/>
  </sequence>
</complexType>
<element name=”para” type=”paraType”/>
```

В результате:

```
<para>
  …
    <strong>…</strong>
  …
</para>
```

При построении сложного типа определяется «модель содержимого» (content model). Она, если не считать атрибутов, строится из так называемых групп (model Groups), к числу которых относятся sequence и choice. Существует еще одна специальная группа all. Ее особенность – она может содержать лишь элементы, а не другие группы. При этом эти элементы не могут повторяться более одного раза, то есть атрибуты minOccurs и maxOccurs могут принимать значения 0 или 1.

Группа all может быть группой только самого верхнего уровня в content model. Элементы, входящие в эту группу могут идти в произвольном порядке и при соответствующих ограничениях могут быть необязательными.

При описании самих групп (all, sequence, choice) можно использовать ограничение повторяемости – minOccurs и maxOccurs.

В языке схем существует возможность повторного использования фрагментов схемы за счет определения групп элементов или атрибутов.

Группы элементов описываются элементом group, а группы атрибутов – attributeGroup. Эти группы можно использовать при описании сложных типов, ссылаясь на них по имени. При этом используются те же самые элементы group и attributeGroup с атрибутом ref.

В этом смысле группы подобны параметризованным сущностям DTD.

```
<group name=”userElements”>
  <sequence>
    <element name=”login” type=”string”/>
    <element name=”password” type=”hexBinary”/>
  </sequence>
</group>
<attributeGroup name “userAttrs”>
  <attribute name=”id” type=”ID”/>
</attributeGroup>
<complexType name=”customerType”>
  <sequence>
    <group ref=”userElements”/>
    <element name=”firstName” type=”string”/>
      … … …
  </sequence>
  <attributeGroup ref=”userAttrs”/>
  <attribute name=”lastLogin” type=”dataTime”/>
</complexType>
```

ГРУППЫ ПОДСТАНОВКИ (SUBSTITUTION GROUP)

Идея состоит в том, чтобы вместо некоторого элемента, который называется головным (head element) можно использовать любой из других элементов, входящих в substitution group. При этом эти элементы должны иметь тип, совпадающий с типом head или унаследованный от него.

Принадлежность элемента группе подстановки описывается атрибутом substitutionGroup, значение которого указывает имя головного элемента.

Использование элементов из групп подстановки вместо головного не является обязательным, то есть можно использовать и головной элемент и любой из группы подстановки вместо него.

```
<element name=”a” type=”string”/>
  … … …
<complexType name=”…”>
  <sequence>
    <element name=”b” substitutionGroup=”a”/>
  </sequence>
</complexType>
```

Головной элемент должен быть глобальным!!!

ИМПОРТ ТИПОВ СХЕМ

Существует возможность описания типов в одной схеме, а использование – в другой. При этом целевые пространства имен обеих схем могут быть различны. Для этого предусмотрен элемент import, который должен быть первым потомком элемента schema и их может быть несколько.

При импорте указывается пространство имен импортируемой схемы атрибутом namespace. Существует также механизм простого включения фрагментов схемы из других источников с помощью элемента include, который указывает размещение этого источника атрибутом schemaLocation.

Так как при простом включении нельзя идентифицировать пространство имен, то считается, что включаемый фрагмент относится к тому же пространству имен, что и основная схема.

В языке схем базовым для абсолютно всех типов, в том числе для встроенных, является тип anyType. Базовый для всех простых типов является anySimpleType, производный от anyType. Базовый для всех комплексных – anyComplexType. Эти базовые типы не накладывают никаких ограничений на свои значения.

Существует также возможность гибко управлять ограничениями при описании конкретных сложных типов с помощью элементов any и anyAttribute. Any используется для обозначения того, что может быть использован любой элемент из любого пространства имен, которое указывается атрибутом namespace и может содержать список идентификаторов и специальные значения: ##any – по умолчанию для атрибута namespace, ##targetNamespace, ##local, ##other.

Способ обработки XML-анализатором таких элементов указывается атрибутом processContents, который может принимать значения: skip, strict, lax. Если strict, то XML-процессор такие элементы проверяет на соответствие схеме, соответствующего пространства имен. Если skip, то соответствующие элементы не проверяются на соответствие схеме. Если lax, то проверяются на соответствие только те элементы, для которых схема может быть извлечена.

Специальные значения для namespace: any – любой идентификатор пространства имен, local – элементы локальны, не принадлежат никакому пространству имен, other – элементы из пространства имен за пределами целевого, targetNamespace – элементы принадлежат целевому пространству имен.

anyAttribute – аналогичен any, но используется только для атрибутов, атрибуты namespace и processContents так же имеют место.

```
<complexType name=”text”>
  <complexContent mixed=”true”>
    <restriction base=”anyType”>
      <sequence>
        <any minOccurs=”0” maxOccurs=”unbounded” processContents=”lax”/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

Для включения в схему произвольного фрагмента, соответствующего другой схеме.

```
<complexType name=”…”>
  <sequence>
    <element name=”…” type=”…”/>
      … … …
    <element name=”htmlExample”>
      <complexType>
        <sequence>
          <any namespace=”http:www.w3.org/1999/xhtml” minOccurs=”1”
maxOccurs=”unbounded” processContents=skip”/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

ОГРАНИЧЕНИЕ УНИКАЛЬНОСТИ. КЛЮЧИ И ССЫЛКИ НА НИХ.

Ограничение уникальности позволяет формально описать требование к уникальности элементов или их атрибутов в некоторых границах. Условие уникальности описывается внутри элемента element элементом unique. Имя условия уникальности указывается атрибутом name. Область, к которой относится данное условие уникальности описывается элементом (вложенным) selector, у которого есть атрибут xpath. Значение xpath есть XPath выражение, ограничивающее область применения условия уникальности.

Что контролировать на уникальность описывается элементами field также в виде XPath выражения, указанными в виде атрибутов xpath.

XPath выражения ограничивают область действия, указывая путь к вершине дерева элементов, начиная с которых она (область) распространяется на дочерние элементы или указывая имена элементов, содержащихся в других элементах.

```
<complexType>
  <sequence>
    <element name=”lineItem” type=”lineItemType” maxOccurs=”unbounded”/>
  </sequence>
</complexType>
<complexType name=”lineItemType”>
  <sequence>
    <element name=”sku” type=”skuType”/>
    <element name=”quantity” type=”decimal”/>
    <element name=”price” type=”decimal”/>
  </sequence>
</complexType>
<element name=”lineItems” type=”lineItemsType”/>
  <unique>
    <selector xpath=”.”/>
    <field xpath=”lineItem/sku”/> (в пределах lineItem контролируем уникальность элементов sku, вложенных в lineItem)
  </unique>
</element>
```

Если изменить описание lineItemType, заменив вложенные элементы атрибутами, то условие уникальности приняло бы вид:

```
<unique name=”uniqueSku”>
  <selector xpath=”lineItem”/>
  <field xpath=”@sku”/>
</unique>
```

Использование ключей (key) и ссылок на них (keyRef) позволяет описать требования к соответствию значений одних элементов или атрибутов другим элементам или атрибутам из некоторой области. Те элементы или атрибуты в пределах той области должны быть уникальны, что позволяет однозначно на них ссылаться. Ссылки уникальными быть не обязаны.

Рассмотрим пример документа:

```
<openOrderInfo>
  <products>
    <product sku=”123”>
      <model>…</model>
      <vendor>…</vendor>
      <category>…</category>
    </product>
    … … …
  <products>
  <order>
    <shipping-address>
      … … …
    </shipping-address>
    <lineItems>
      <lineItem sku=”123” quantity=”50” price=”0.2”/>
```

```
    </lineItems>
  </order>
</openOrderInfo>
```

В этом примере значение атрибута sku элементов lineItem должно соответствовать какому-либо значению атрибута sku элемента product. Причем может быть несколько одинаковых ссылок на один и тот же продукт, не в пределах одного заказа.

Соответствующий фрагмент схемы (наиболее важные моменты):

```
<element name=”products” type=”productsType”>
  <key name=”keyProducts”>
    <selector xpath=”product”/>
    <field xpath=”@sku”/>
  </key>
</element>
<element name=”lineItems” type=”lineItemsType”/>
  <unique name=”uniqueSku”>
    <selector xpath=”lineItem”/>
    <field xpath=”@sku”/>
  </unique>
  <keyref name=”productRef” refer=”keyProduct”>
    <selector xpath=”lineItem”/>
    <field xpath=”@sku”/>
  </keyref>
</element>
```

В элементе keyRef атрибут refer указывает имя ключа, определенного элементом key.

Замечание. Условие уникальности и ключи могут быть составными, то есть определяться по сочетанию значений элементов или атрибутов. Для этого необходимо использовать несколько элементов field в элементе unique или key. Ссылки на составные ключи так же являются составными.

ССЫЛКИ НА СХЕМЫ

Ссылку на схему можно внедрить в XML-документ. Для этого предусмотрены атрибуты: xsi:schemaLocation, xsi:noNamespaceSchemaLocation, которые можно использовать из любого элемента. Первый атрибут – для использования схем, у которых есть целевое пространство имен. Его значение – список пар «идентификатор_пространства_имен – физический URI схемы». Второй атрибут – для использования схем без целевого пространства имен. Его значения – физические URI схемы.

Для элемента include языка схем обязательный атрибут schemaLocation указывает физический URI включаемой схемы.

Для элемента import языка схем существует необязательные атрибуты namespace и schemaLocation. Если импортируются типы из некоторого целевого пространства имен, используется атрибут namespace, указывающий его идентификатор, и schemaLocation, указывающий физический URI для соответствующей схемы. Если импортируются типы, с которыми не связано никакое пространство имен, атрибут namespace не используется.

Лекция 2 (дополнение 1). Шаблоны проектирования XML-схем

Глобальный элемент	Создавать типы	Название шаблона	Описание	Достоинства	Недостатки
1	Да	Venetian Blind (Жалюзи)	Объявления элементов вложены в единственное глобальное объявление, в них используются именованные сложные типы и группы элементов. Сложные типы и группы элементов могут использоваться во всей схеме. В глобальном пространстве имен должен быть объявлен только корневой элемент.	- единственный корневой элемент; - обеспечивает повторное использование всех типов и единственного глобального элемента; - позволяет использовать несколько файлов.	- нарушение инкапсуляции.
1	Нет	Russian Doll (Матрешка)	В отличие от шаблона Жалюзи объявления элементов могут использоваться только один раз.	- единственный корневой элемент; - может уменьшить сложность, связанную с использованием пространства имен (зависит от атрибута схемы elementFormDefault).	- допускает повторное использование либо всех элементов, либо ни одного; - схема может состоять только из одного файла.
Много	Да	Garden of Eden (Райский сад)	Комбинация шаблонов Жалюзи и Салями. Все элементы и типы определяются в глобальном пространстве имен, в объявлении сложного типа на элементы даются ссылки.	- обеспечивает повторное использование всех типов и элементов; - позволяет использовать несколько файлов.	- нарушение инкапсуляции; - много потенциальных корневых элементов; - схема сложна для чтения и понимания.
Много	Нет	Salami Slice (Салями)	Все элементы являются глобальными (определяются в глобальном пространстве имен). Объявления элементов не вкладываются друг в друга и используются во всей схеме.	- обеспечивает повторное использование всех элементов; - поддерживает использование элементов из других документов.	- много потенциальных корневых элементов.

Моделирование XML-словарей с помощью UML, часть II

Автор: Dave Carlson, 19 сентября 2001

http://www.raleigh.ru/a/pub/2002/uml.html?page=2

Отображение UML-моделей в схемы XML

Именно это является ключевым моментом использования UML при разработке XML-схем. Основной целью, определяющей спецификацию такого отображения, является обеспечение его достаточной гибкостью, чтобы покрыть большинство требований к разработке схем, но при этом сохранить "мягкий" переход от концептуальной модели словаря к его детальной проработке и последующей реализации.

Родственная цель — создать условия для автоматической генерации корректной XML-схемы из любой диаграммы классов UML, даже если тот, кто моделирует не знаком с синтаксисом XML-схем. Наличие такой возможности допускает быстрый процесс разработки и поддерживает многократное использование моделей словарей для различных языков реализации или окружений, поскольку сама модель не слишком привязана непосредственно к XML.

Отметьте, пожалуйста, что примеры схем в этой статье не полностью сопоставимы с соответствующим примером в XML Schema Primer. Тем не менее, следующие фрагменты схем являются корректными интерпретациями концептуальной модели. Третья статья этого цикла продолжит процесс усовершенствования вплоть до логического завершения, когда результирующая схема сможет подтвердить пример из XSD Primer.

Концептуальная модель для нашего словаря заказов показана на рисунке 1, она воспроизведена из первой статьи с очень небольшими изменениями. Мы разобъем эту диаграмму на главные структуры и отобразим каждую ее часть на язык W3C XML Schema. Я отмечу несколько случаев, где возможны другие альтернативы, и также укажу, где результирующая схема отличается от примера из XSD Primer.

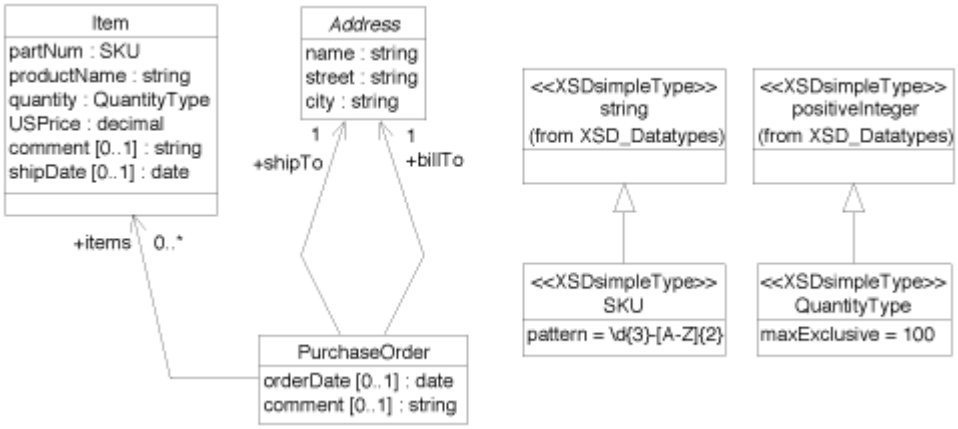


Рис. 1: *Концептуальная модель словаря заказов покупок*

Класс и атрибут

Класс в UML определяет сложную структуру данных (и соответствующее поведение), которая по умолчанию отображается в `complexType` в XSD. На первом этапе класс Purchase Order и его UML атрибуты создают следующее определение XML Schema:

```
<xs:complexType name="PurchaseOrder">
  <xs:all>
    <xs:element name="orderDate" type="xs:date"
                minOccurs="0" maxOccurs="1"/>
    <xs:element name="comment" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
  </xs:all>
</xs:complexType>
```

Атрибуты в UML-классе не имеют заданного порядком, поэтому для создания неупорядоченной группы используется элемент XSD `<xs:all>`. Кроме того, класс UML создает различные пространства имен для имен своих атрибутов (например два класса могут содержать атрибуты, имеющие одинаковые имена), так что они создаются в схеме как локальные определения элементов. Для более широкого ознакомления с этой темой смотрите A New Kind of Namespace. Оба атрибута в нашей модели UML не обязательны, что показано на Рисунке 1 как `[0..1]`. Эти значения отображаются в атрибуты minOccurs и maxOccurs в XSD. UML-атрибуты определялись с помощью простейших типов данных из спецификации XSD, так что они записываются прямо в результирующую схему с использованием подходящего префикса пространства имен. Если же в модели UML используются другие типы данных, то для использования этих типов в схеме может быть создана библиотека типов XSD. Например, я создал библиотеку типов XSD для простейших типов языка Java и для простейших классов Java таких, как Date, String, Boolean, и т.д.

Полезно заметить, что элемент верхнего уровня автоматически создается в схеме для каждого complexType. По умолчанию имя для этого элемента такое же как и имя класса, — это возможно в W3C XML Shema, поскольку используются различные пространства имен внутри одной схемы для complexType и для элементов верхнего уровня. Для PurchaseOrder элемент верхнего уровня схемы создается следующим образом:

```
<xs:element name="PurchaseOrder" type="PurchaseOrder"/>
```

Если вы обратитесь к примеру в XSD Primer, вы увидите, что `orderDate` моделируется как XML-атрибут, а не как дочерний элемент PurchaseOrder. Кроме того там используется группирующий элемент `<sequence>` вместо `<all>`. И, в-третьих, элемент верхнего уровня определялся в Primer с помощью строчной первой буквы, то есть purchaseOrder (его часто называют форматом "lower camel case"). Здесь я адресую вас к третьей статье, где используется UML profile /*профили*/ для расширения отображений на схемы XML.

Ассоциация

Тип PurchaseOrder в нашей модели определялся не только при помощи атрибутов UML, но также своими ассоциациями с другими классами в модели. На Рисунке 1 показаны три ассоциации, в которых участвует PurchaseOrder. У каждой ассоциации есть роль и мощность, которые точно определяют отношение с целевым классом. Эти ассоциации добавляются к группе complexType в XSD вместе с элементами, созданными из атрибутов UML.

```
<xs:complexType name="PurchaseOrder">
  <xs:all>
    <xs:element name="orderDate" type="xs:date"
                minOccurs="0" maxOccurs="1"/>
    <xs:element name="comment" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
    <xs:element name="shipTo">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Address"/>
        </xs:sequence>
      </xs:complexType>
```

```
    </xs:element>
    <xs:element name="billTo">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Address"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="items" minOccurs="0" maxOccurs="1">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Item"
                      minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:all>
</xs:complexType>
```

Поскольку UML-атрибуты для `orderDate` и `comment` имеют простейшие типы данных, схема включает их в себя в качестве содержания элемента. Однако, по умолчанию при отображении для ассоциаций в XSD создаются покрывающие элементы в соответствии с именем роли в UML. Кроме того эти элементы содержат требования к ассоциированному классу, к которому схема обращается при помощи элемента верхнего уровня, созданного для каждого complexType.

Если вы хотите создать W3C XML Schema, используя модель содержания <all>, то покрывающий элемент будет необходим всегда, когда ассоциированный класс имеет более одного случая. Причина в том, что <all> может быть использован только, когда внутренние элементы имеют мощность либо [0..1], либо [1..1]. Так, когда создается покрывающий элемент для ассоциации с `Item`, элемент, называющийся `item`, может иметь либо ноль, либо одну реализацию, которая, в свою очередь, содержит ноль, или более элементов `Item` внутри себя.

Разница между схемой, созданной с помощью правил по умолчанию из UML, и схемой, включенной в XSD Primer в том, что роли shipTo и billTo в Primer содержат прямое указание на адрес, без обращения к элементам ассоциированного класса. Другими словами, дочерние элементы для имени, улицы, города и т.д. содержатся прямо в shipTo и billTo. Этот альтернативный подход мы еще раз дополнительно осветим в третьей статье.

Тип данных, определенный пользователем

По умолчанию при отображении в XSD должно генерироваться определения complexType для SKU и QuantityType, но мы хотим, чтобы это происходило, как определение пользователем простых типов данных в XML-схеме. Этого достаточно просто добиться, добавив UML-стереотип, который показан как <<XSDsimpleType>> на Рисунке 1, в каждый из этих двух классов. Эта возможность включения стереотипов является неотъемлемой частью стандарта UML и используется для формального определения дополнительных характеристик модели, которые, как правило, уникальны для каждой конкретной области, в нашем случае, уникальны для проектирования XML-схемы.

Используя стереотип, генератор схемы знает, как создать следующее определение для SKU:

```
<xs:simpleType name="SKU">
  <xs:annotation>
    <xs:documentation>Stock Keeping Unit, a code
         for identifying products</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3}-[A-Z]{2}"/>
```

```
  </xs:restriction>
</xs:simpleType>
```

Кроме того, UML-модель может включать документацию для каждого элемента модели. Такая документация приводится в определении XML-схемы, как показано в этом примере. Обобщенное отношение UML показывает, какие существующие простые типы данных должны быть использованы в качестве основы для определяемого нового типа. В конце концов, атрибут pattern в SKU, отображенный в формат XSD, вынужден принять в качестве значения строку SKU.

Второй модуль в определении схемы purchase order представляет повторно используемый набор формальных определений для адресов (это показано на рисунке 2). Эти определения позаимствованы из раздела 4.1. в XSD Primer. В нашей модели успользуются два дополнительных конструктора схемы, в добавление к использованным при составлении схемы на рисунке 1.

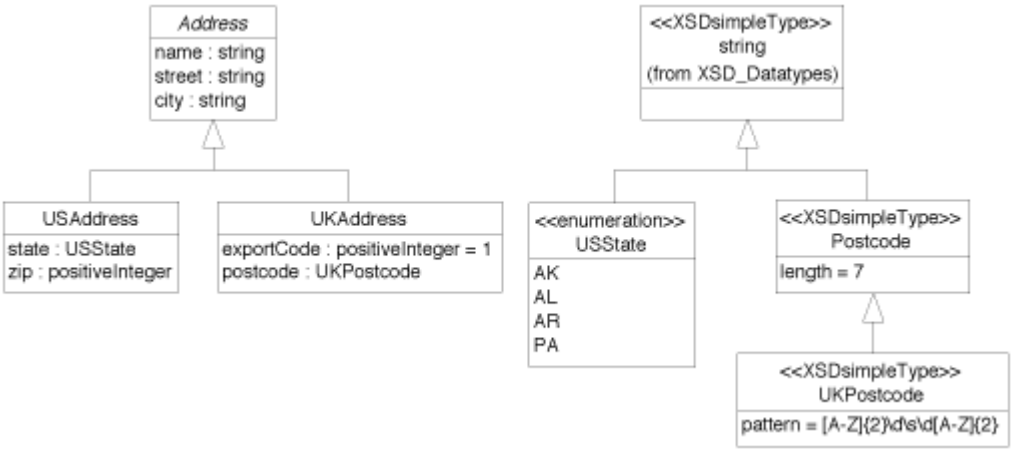


Рис. 2: *Компонента схемы Modularized Address*

Обобщение

Обобщение — это фундаментальное и распространенное понятие в объектно-ориентированном анализе и проектировании. Специализированные подклассы наследуют атрибуты и ассоциации от всех родительских классов. Это легко представляется в W3C XML Schema, хотя требует более изощренных механизмов при использовании других языков XML-схем.

На Рисунке 2 класс Address, обозначенный курсивом, использующимся в UML для обозначения абстрактного класса, предназначается только для определения других специализированных классов. Следуя тем же правилам, использующимся по умолчанию для PurchaseOrder, определения complexType для Address и USAddress создаются следующим образом:

```
<xs:element name="Address" type="Address" abstract="true"/>
<xs:complexType name="Address" abstract="true">
  <xs:all>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
  </xs:all>
</xs:complexType>

<xs:element name="USAddress" type="USAddress"
            substitutionGroup="Address"/>
<xs:complexType name="USAddress">
  <xs:complexContent>
    <xs:extension base="Address">
      <xs:all>
```

```
        <xs:element name="state" type="USState"/>
        <xs:element name="zip" type="xs:positiveInteger"/>
      </xs:all>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Здесь три отличия от прошлых примеров. Во-первых, элемент верхнего уровня и определение complexType для Address включают XSD-атрибут `abstract="true"`. Во-вторых, элемент USAddress включает в себя `substitutionGroup="Address"`; это означает, что каждый раз, когда элемент Address запрашивается в качестве элемента содержания, USAddress может быть заменен. Таким образом, мы можем использовать USAdress (или, аналогично, UKAdress), как содержание shipTo и billTo в PurchaseOrder.

В-третьих, определение complexType для USAddress распространяется из основного complexType для Address. В этом и заключается очень серьёзное различие между тем, как эти структуры наследования интерпретируются в UML и, как это происходит в XSD. В UML порядок атрибутов и ассоциаций внутри класса не имеет значения, а в подклассе характерные особенности, наследуемые из родительских классов, свободно перемешиваются с локально определенными атрибутами и ассоциациями. В XSD наследуемые элементы трактуются как группа, так что три элемента, унаследованные из Address представляют из себя неупорядоченную группу в USAddress, и следуют один за другим рядом с другой неупорядоченной группой из двух элементов, определенной в USAddress. Вы не можете определить пятиэлементную группу, когда один или более элемент унаследован.

Перечисляемый тип данных

Элемент `state` USAddress приписывает определение простого типа для `USState`. Это определение производится из UML-перечисления. На Рисунке 2 USState показан вместе со стереотипом <<enumeration>>, что извещает генератора схемы о необходимости создать XSD-перечисления для каждого атрибута, определенного для данного класса. Перечисляемый тип в XSD, это всего лишь специализированный вид определений simpleTipe, так что он должен еще указывать суперкласс в UML, чтобы использоваться как базовый тип в XSD. Получается следующая схема:

```
<xs:simpleType name="USState">
  <xs:restriction base="xs:string">
    <xs:enumeration value="AK"/>
    <xs:enumeration value="AL"/>
    <xs:enumeration value="AR"/>
    <xs:enumeration value="PA"/>
  </xs:restriction>
</xs:simpleType>
```

Заключение

Правила отображения по умолчанию, описанные в этой статье, могут быть использованы для создания полной XML-схемы из любой диаграммы классов UML. Это, может быть, ранее существовавшая модель приложения, которая сейчас должна использоваться в рамках архитектуры веб-сервисов XML, или это может быть модель нового XML-словаря, определяемая в качестве стандарта для обмена данными уровня B2B. В любом случае, схема, полученную с помощью этих правил, обеспечивает удобный способ получения первого приближения, которое уже может быть немедленно использована в первоначальном развертывании приложения, хотя и может потребовать некоторой доработки для соответствия другим архитектурным и проектным требованиям.

Первая статья этого цикла посвящена процессу разработки схемы, в ней подчеркивалось различие между созданием для data-ориентированного и для text-ориентированного

приложения. Правил отображения по умолчанию часто достаточно для data-ориентированных приложений. Фактически, эти правила соответствуют спецификации OMG XML Metadata Interchange (XML) 2.0 для использования XML в качестве формата обмена моделями. Этот подход еще хорошо реализован вместе с новой инициативой OMG по Model Driven Architecture (MDA).

Text-ориентированные схемы, и любая другая, которая может быть кем-нибудь придумана и использована для содержания для веб-порталов, часто должны быть усовершенствованы, чтобы упростить структуру XML-документов. Например, многие проектировщики схем устраняют покрывающие элементы, соответствующие роли ассоциации (но это также предупреждает использование группирующего элемента XSD <all>). Эта доработка, а также многие другие, могут быть определены в модели словаря путем помещения нового параметра по умолчанию для одного пакета UML, который затем применяется ко всем содержащимся в нем классам.

В этой статье мы рассмотрели два примера стереотипов UML, которые были созданы, чтобы отметить специализированное использование класса UML. Более обобщенно, эти стереотипы и ассоциированные с ними значения свойств являются частью UML-профиля для XML-схем, который я вначале разрабатывал в рамках моей книги о моделировании приложений XML. Третья статья этого цикла представляет дополнительные примеры использования других стереотипов для уточнения результирующей схемы. Также я предложу вашему вниманию описание инструментов, которые были нами разработаны. Эти инструменты представляют полный UML-профиль для проектирования схемы и преобразования любой модели класса UML либо в W3C XML Schema, либо в грамматику OASIS RELAX NG.

Советы на будущее

Для того, чтобы облегчить вам применение изложенных идей на практике при выполнении собственных проектов, я предлагаю следующие полезные советы.

- План концептуальной модели ваших рабочих словарей может быть использован в нескольких различных контекстах развертывания, т.е. W3C XML Schema, DTD, реляционной DBMS, Java или EJB, и т.д. Альтернативные UML-профили могут быть использованы для трансформации общей рабочей модели на другую платформу. Однако помните, что полная реализация этой цели лежит за пределами возможностей большинства современных инструментов UML.
- Существующие UML-модели могут быть специализированы под конкретную платформу развертывания, библиотеки и типы данных (Java, NET, и т.д.) Изолируйте модель предметной области от конкретной платформы, чтобы иметь возможность ее повторного использования и дальнейшей разработки XML-схем для обмена данными.
- Используйте совместимые правила моделирования для имен и структуры, как в рамках простого словаря, так и для всех родственных моделей. Например, спецификация архитектуры FpML предусматривает нормативы для написания DTD, что делает не сложным переход к UML-моделям или любой другой объектно-ориентированной среде.

Лекция 3. Язык XPath

ЯЗЫК XPATH

Технология XPath реализует возможности, которые используются в других XML-технологиях. Например, XPath выражение используется для описания условий уникальности ключей и ссылок на них в языке XML-схем. Так же XPath применяется в составе конструкций XSLT, которые описывают процедуру преобразований исходного XML-документа к документу другого вида. В общем можно сказать, что технология XPath используется не сама по себе, а совместно с другими XML-технологиями.

Основная задача языка XPath – обеспечить средства адресации различных частей XML-документа и базовые средства для манипуляции строковыми, цифровыми и булевскими значениями. При этом используется компактный не-XML синтаксис с тем, чтобы была возможность использовать XPath-выражение в составе URI или как значение XML атрибутов.

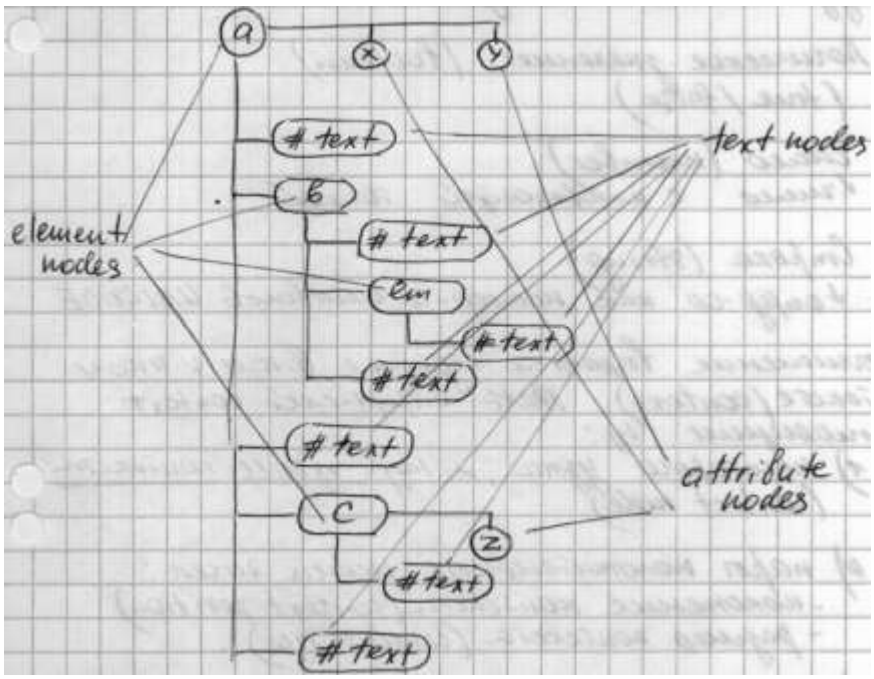
Кроме адресации некоторые виды XPath-выражений используются для проверки соответствия узла дерева XML-документа некоторому шаблону (pattern matching). Такое применение XPath находит в технологии XSLT.

Для вычисления XPath-выражений XML-документ представляется как дерево узлов. Существует несколько разных типов узлов: узлы, соответствующие элементам (element nodes); узлы, соответствующие атрибутам (attribute nodes), узлы, соответствующие тексту (text nodes).

Пример. XML-документ:

```
<a x=”…” y=”…”>
  <b>
    Hallo,<em>Vasya</em>!
  </b>
  <c z=”…”>
    Good Bye!
  </c>
</a>
```

Дерево, которое соответствует XML-документу (DOM):



Детально структура этого дерева закреплена спецификацией W3C Document Object Model (DOM), которая определяет, как XML-документ может быть представлен в виде дерева объектов, а также набор интерфейсов для работы с этими объектами.

XPath определяет способ получения строкового значения для узла любого типа.

XPath совместим со спецификацией XML Namespaces, то есть имя узла состоит из пары: локальное имя (local part) и необязательного идентификатора пространства имен (namespace URI).

Основная синтаксическая конструкция XPath – выражение. Результат вычисления выражения – объект одного из 4-х базовых типов:

1. множество узлов (node-set) – неупорядоченная коллекция узлов без дубликатов.

2. логическое значение (boolean) – true или false.

3. число (number) – число с плавающей точкой.

4. строка (string) – определяется как последовательность символов UNICODE

Вычисление выражения ведется в том или ином контексте (context). XPath определяет контекст состоящим из:

1. некоторого узла, который называется контекстным узлом (context node)

2. пара положительных чисел: положение контекста (context position), размер контекста (context size)

3. множества переменных (variable bindings)

4. библиотеки функций (function library)

5. множества объявлений пространств имен (namespace declaration) в области видимости выражения

Позиция контекста всегда меньше или равна размеру контекста. Множество переменных определяет соответствие между именами переменных и их значениями. Значением переменной является объект одного из 4-х базовых типов или какого-то другого типа, неопределенного спецификацией XPath. Дополнительные типы могут определяться технологией, совместно с которой используется XPath.

Библиотека функций определяет соответствие между именами функций и самими функциями. Каждая функция принимает 0 или более аргументов и возвращает единственный результат. Спецификация XPath определяет набор функций, которые формируют core function library – она должна поддерживаться любой реализацией XPath. Функции из core function library оперируют только 4-мя базовыми типами.

Объявления пространств имен определяют соответствия между префиксами (prefixes) и идентификаторами (namespaces URI).

Множество переменных, библиотека функций и объявления пространств имен, используемые для вычисления подвыражений, всегда соответствуют тем, что используются для содержащего его выражения, а контекстный узел, позиция и размер контекста могут изменяться.

XPath-выражения могут использоваться как значения XML-атрибутов. Если XPath-выражение содержит символ «<», он должен заменяться esc последовательностью <. Кроме того, содержащиеся в XPath-выражении кавычки и апострофы, используемые как ограничители строковых выражений, могут заменяться соответственно на ", ', если такие же символы используются как ограничители значения XML атрибута.

Наиболее важным видом XPath-выражений являются пути (location paths). Путь выбирает множество узлов относительно контекстного узла. Результат вычисления такого выражения – node-set, содержащий выбранные узлы. Пути могут содержать дополнительные выражения, используемые для фильтрации полученного множества выбранных узлов.

LOCATION PATHS

Для Location Paths предусмотрены полный и сокращенный синтаксисы. Сокращенный синтаксис (abbreviated syntax) вводит упрощенную запись для более часто используемых частных случаев.

Существует два вида путей: абсолютный и относительный. Относительные пути (relative location paths) состоят из одного или более шагов (location steps), разделяемых символом «/». Шаги обрабатываются последовательно слева направо. Каждый шаг выбирает определенное множество узлов относительно контекстного. Каждая вершина из множества, которая является результатом очередного шага, используется как контекстная для следующего шага. Все множества узлов, полученные таким образом на следующем шаге, объединяются в единое множество, которое является результатом этого шага.

Например, child::div/child::p – это выражение выбирает все элементы p, у которых родитель – элемент div и которые является «внуками» контекстного узла (потомками второго уровня контекстного узла).

#LOCATION STEPS

Location steps состоит из 3-х частей:

1. ось (axis), которая определяет отношение в дереве между контекстными location steps.

2. условие для узлов (node test) определяет тип узлов и их полное имя (expanded name) с учетом пространства имен, которое выбирается данным location steps.

3. ноль или более предикатов (predicates). Предикаты задают дополнительные ограничения, включаемых в результирующий node set.

Синтаксис location steps: имя_оси :: node_test[predicate]…

Например: child :: para[position()=1] – выбираем все 1-ые дочерние элементы с именем para (1-ый элемент)

При вычислении location step предикаты вычисляются в последнюю очередь, ограничивая множество на основе пары axis::node_test. Предикаты вычисляются последовательно слева направо.

Axes (ОСИ)

Доступны следующие оси:

- child – все непосредственные потомки контекстного узла;
- descendant – все любые потомки контекстного узла;
- parent – непосредственный родитель контекстного узла;
- ancestor – любой предок контекстного узла;
- following-sibling – все сестринские узлы относительно контекстного, которые расположены после него;
- preceding-sibling - все сестринские узлы относительно контекстного, которые расположены перед ним.

Если контекстный узел является attribute node или namespace node, то последние две оси – пустые. У всех following и preceding sibling parent такой же как и у context node.

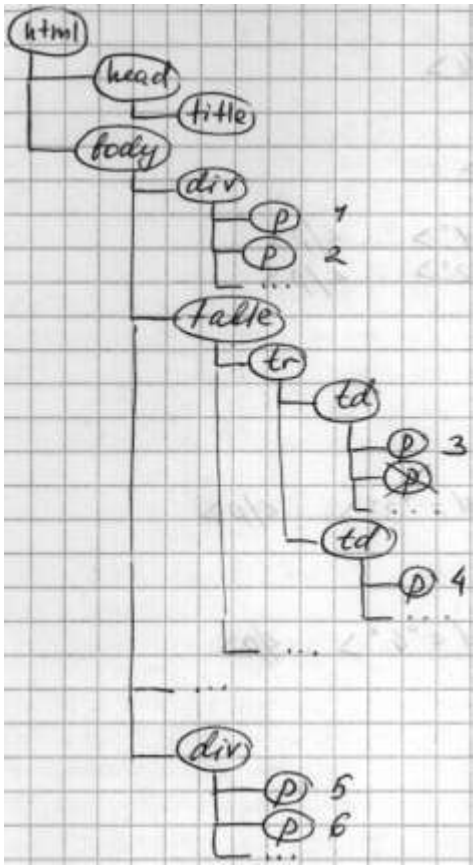
Following, preceding – все узлы, которые следуют после или до context node в документе. Attribute – содержит все атрибуты контекстного узла. Ось не пустая только для тех context node, которые являются element node. Namespace – содержит все namespaces nodes контекстного узла. Namespace nodes соответствует объявлениям и префиксов для namespace URIs. Они не пусты только для element node. Self – соответствует самому context node. Descendant-or-self, ancestor-or-self – объединения указанные в названии осей.

Оси ancestor, descendant, following, preceding, self делят документ на пять разделов, которые вместе содержат все элементы документа, но при этом не пересекаются.

Пример. Рассмотрим дерево из предыдущего примера. В качестве context node выберем узел table, тогда ось self содержит следующий узел – table. Ось ancestor – body, html. Ось descendant – tr, td, p, td, p.

```
<html>
  <head>
    <title>…</title>
  </head>
  <body>
    <h1>…</h1>
    <div>
      <p id=”1”>…</p>
      <p id=”2”>…</p>
      … … …
    </div>
      … … …
    <table>
      <tr>
        <td>
          <p id=”3”>…</p>
            … … …
        </td>
        <td>
          <p id=”4”>…</p>
            … … …
        </td>
         … … …
      </tr>
       … … …
    </table>
     … … …
    <div>
      <p id=”5”>…</p>
      <p id=”6”>…</p>
      … … …
    </div>
  </body>
</html>
```

Если не учитывать текстовые элементы и атрибуты, то дерево этого документа (упрощенное) будет выглядеть так



Если context node для приведенного XPath выражения будет элемент body, то результат выражения – это node set, содержащий элементы p с id=1,2,5,6. Элементы p с id=3,4 в него не войдут.

Абсолютные пути (absolute location paths) начинаются с символа /, после которого могут идти (необязательно) relative location paths. Абсолютный путь / соответствует корневому элементу документа, который является контекстным узлом. Если после / идет относительный путь, то этот путь вычисляется с использование корневого элемента документа как контекстного узла. Ось following – div, p, p. Ось preceding – p, p, div, title, head.

У каждой оси есть базовый тип узла, который она содержит:

1. для оси attribute – attribute node

2. для оси namespace – namespace node

3. для всех других осей – element node

Условие для узлов (node test), заданное квалифицированным именем, дает истинный результат, для узлов базового типа, у которых полное имя (expanded name) равно полному имени, указанному в виде квалифицированного имени в условии. Преобразование квалифицированного имени в полное выполняется с учетом объявления пространств имен (namespace declarations) контекста выражения.

Префикс квалифицированного имени при этом значения не имеет. Если условие не выполняется ни для одного из узлов оси, результат – пустой node set. Если условие задано в виде * - в результирующий node set включаются все узлы оси. Условие может быть

задано в виде: prefix:*. Оно дает истину для всех узлов оси базового типа, полное имя которого имеет такой же идентификатор пространства имен, второй соответствует указанному в условии префиксу без учета их локальных имен. Условие задается в виде text(), дает истину для любого текстового узла оси. Аналогично условие comment() дает истину для любого комментария (comment node). Условие в виде node() выбирает узел любого типа. Условие processing-instruction() дает истину для любых инструкций обработки. Это условие может иметь аргумент, задающий имя инструкции обработки, тогда условие дает истину лишь для инструкций обработки с соответствующим именем.

ПРЕДИКАТЫ

Предикаты – любое выражение, результат вычисления которого может быть преобразован к логическому значению. Преобразование выполняется по тем же правилам, что и для функции boolean. Если значение предиката числовое, то результат считается истинным при соответствии этого числа позиции (context position) узла.

Любая ось является либо прямой, либо обратной. Оси, которые содержат контекстный узел и узлы, следующие за ним в документе, называют прямыми. Оси, содержащие узлы предшествующие контекстному, и сам контекстный узел – обратные. Forward axis – прямые оси. К ним относятся оси: attribute, child, following, following-sibling, namespace, descendant, descendant-or-self. Reverse axis: ancestor, ancestor-or=self, parent, preceding, preceding-sibling. Для оси self понятия forward и reverse не имеют смысла.

PROXIMITY POSITION

Proximity position – характеризует положение узла, входящего в node set, в документе. Используется прямой порядок следования узлов для прямых осей и обратный для обратных.

Первая позиция начинается в единицы (1). Например, preceding-sibling::p[position()=1] – выбирает ближайший предыдущий сестринский узел P для p-контекстного (то есть на том же уровне вложенности), так как preceding-sibling – обратная ось. Following-sibling::p[position()=1] – следующий узел p за контекстным узлом на том же уровне вложенности.

Примеры Location Paths. Child::para – выбирает все дочерние узлы с именем para. Chilp::* - выбирает все дочерние узлы. Child::node() - выбирает все дочерние узлы любого типа. Attribute::name – выбирает атрибут с именем name. Attribute::* - выбирает все атрибуты контекстного узла. Child::chapter/descendant::para – выбирает все элементы с именем para, которые содержатся внутри дочерних элементов chapter. / - соответствует корню документа. Child::para[position()=last()] – возвращает последний из дочерних элементов para. Child::chapter[child::title=”introduction”] – у контекстного узла выбирает все chapter, а у них – все дочерние title, у которых текст = introduction. Child::para[attribute::id=”123”] – проверка по атрибутам, выбирает дочерние элементы para, у которых атрибут с именем id=123.

СОКРАЩЕННЫЙ СИНТАКСИС ДЛЯ LOCATION PATHS

child:: - подразумевается по умолчанию, то есть ось child является осью по умолчанию. Child::div/child::para соответствует div/para.

Для attribute:: предусмотрено сокращение @: child::p[attribute::id="123"] соответствует p[@id="123"]

Для |descendant-or-self::node()/ предусмотрено сокращение //. При этом //para[1] (те элементы, которые являются 1-ми потомками своего родителя) не эквивалентно /descendant::para[1] – выбирает 1-го потомка (отсчитывая от корня) любого уровня (то есть 1-ый дочерний элемент para).

Для self::node() существует сокращение «.» (то есть текущий узел): .//para (любой непосредственный потомок с именем para относительно текущего элемента) эквивалентно self::node()/descendant-or-self::node()/child::para.

Parent::node() заменяется на «..»: ../title эквивалентно parent::node()/child::title.

ВЫРАЖЕНИЯ

Location Paths лишь один из видов XPath выражений. Выражение конструируется из литеральных значений, вызовов функций, ссылок на переменные, путей и скобочных выражений.

При этом результаты вычисления подвыражений, дающих node set, можно объединять в один node set операцией |.

Язык XPath определяет набор арифметических операций: +б –б *б div (деление), mod; логических операций: and, or; операции сравнения: =, !=, <=, <, >=, >; ссылки на переменные: $квалифицированное_имя.

Предусмотрены строковые и числовые литералы. Строковые ограничиваются "" или ''. Для числовых предусмотрена только десятичная форма записи.

Для числовых литералов предусмотрены специальные значения: NaN (not a number), положительный и отрицательный ноль, «+» и «-» бесконечность.

Функции для работы с node set:

1. last() – возвращает размер контекста (context size);

2. position() – возвращает context position;

3. count (node set) – возвращает размер node set, переданного как параметр.

Функции для работы со строками:

1. string(object) – преобразует объект в строку, если объект – node set, возвращается строковое значение узла, которое в документе является самым первым (из всех элементов node set). Число преобразуется в строковое значение следующим образом: NaN – 'NaN', ±0 – '0', +? - 'infinity', -? - '-infinity'. Целые числа преобразуются в соответствующие числа без десятичной точки и ведущего плюса. Остальные числа преобразуются в их

строковое изображение без ведущего плюса в виде: . . Булевские значения – ‘true’ или ‘false’.

2. concat – соединяет строковые аргументы в одну строку; аргументов может быть два или более.

3. stars-with – возвращает true, если строка, указанная первым элементом начинается с подстроки, указанной вторым элементом.

4. contains - возвращает true, если строка, указанная первым элементом содержит подстроку, указанной вторым элементом.

5. substring-before – возвращает фрагмент строки, указанной первым аргументом, сначала и до первого вхождения подстроки, указанной вторым аргументом; возвращается пустая строка, если ни одного вхождения не найдено.

6. substring-after - возвращает фрагмент строки, указанной первым аргументом, после первого вхождения подстроки и до конца строки; возвращается пустая строка, если ни одного вхождения не найдено.

7. substring – возвращает подстроку строки, указанной первым аргументом; позиция первого символа подстроки указывается вторым аргументом, а ее длина – третьим. Нумерация позиций начинается с единицы. Третий аргумент необязателен и если он не указан, возвращается подстрока с указанием позиции и до конца троки.

8. string-length – возвращает длину строки; аргумент необязателен. Подразумевается строковое значение контекстного узла

9. normalize-space – выполняет нормализацию строки в терминах XML: удаляются ведущие концевые пробелы, выполняется замена последовательности пробельных символов одиночным пробелом; аргумент необязателен. Если нет аргумента, то подразумевается строковое значение контекстного узла.
10. translate – выполняет замену в троке, указанной первым аргументом символов, указанных в виде строки вторым аргументом, на соответствующие символы, указанных в виде строки третьего аргумента. Соответствие между символами второго и третьего аргумента выполняется по номеру позиции. translate (‘bar’, ‘abc’, ‘ABC’) эквивалентно ‘Bar’. Если символу из второго аргумента не найдено соответствие в третьем аргументе, из исходной строки соответствующий символ удаляется. Если во втором аргументе несколько одинаковых символов, используется позиция самого первого для поиска соответствия с третьем аргументе.

Логические функции:

1. boolean – преобразует объект в булевское значение по следующим правилам: число преобразуется в ‘true’, если оно не ±0б не NaN; node set в ‘true’, если он не пустой; строка в ‘true’, если ее длина больше 0.

2. not – функция выполняется логическую операцию not над аргументом (логическим)

3. true, false – возвращает соответствующие логические значения.

Функции для работы с числами:

1. number – преобразует объект в число по следующим правилам: 1) строка, представляющая число, преобразуется в соответствующее число, любая другая строка – в значение NaN; 2) ‘true’ – в 1, ‘false’ – в 0; 3) node set сначала преобразуется в строку с помощью функции string? Затем строка преобразуется в число по (1).

2. sum – вычисляет сумму значений путем преобразования каждого узла, входящего в node set, указанного как аргумент в число и последующее их суммирование

3. floor – возвращает функцию «пол» от указанного числа

4. ceiling – «потолок» от указанного числа

5. round – округление указанного числа до целого значения

МОДЕЛЬ ДОКУМЕНТА

При вычислении XPath выражений XML-документ представляется в виде дерева узлов. Узлы бывают 7 типов:

1. root node,

2. element node,

3. attribute node,

4. text node,

5. namespace node,

6. comment node,

7. processing-instruction node.

Для каждого узла можно вычислить соответствующее ему строковое значение.

Некоторые типы узлов обладают полным именем (expendant name). При этом идентификатор пространства имен, входящий в состав полного имени может быть null.

Для узлов определяется порядок их следования в документе (document order). root node – всегда первый. element nodes – расположены всегда перед своими детьми. attribute nodes и namespace node – расположены перед детьми элемента. namespace node – перед attribute. Упорядочение namespace и attribute nodes, относящихся в первому элементу, зависит от реализации.

root и element nodes содержат упорядоченный список дочерних узлов, которые никогда не разделяются между несколькими родителями. Любой узел (кроме корневого) имеет строго 1-го родителя (parent).

ROOT NODE

Root node является корнем дерева и в качестве дочерних узлов содержит кроме корневого элемента документа узлы, соответствующие комментариям и инструкциям обработки, из пролога и эпилога XML-документа.

Строковые значения корневого элемента – результат конкатенации строковых значений всех текстовых узлов, являющихся прямыми или косвенными потомками корневого узла в соответствии с порядком в документе. Root node не имеет полного имени.

ELEMENT NODES

Для каждого элемента документа в модели присутствует element node. Полное имя образуется путем расширения префикса соответствующим идентификатором пространства имен. Если элемент указан без префикса и пространства имен по умолчанию нет, то в полном имени идентификатор пространства имен будет null.

Дочерними узлами element node может быть другой element node и comment, processing-instruction и text nodes.

Строковые значения element node определяются так же, как и для root node.

ATTRIBUTE NODES

Каждый element node имеет ассоциированное с ним множество attribute nodes. Для каждого элемента этого множества element node является родителем, но при этом attribute node не относится к дочерним узлам.

Если атрибуту было указано значение по умолчанию, и он отсутствует в документе, то в модели соответствующий attribute node присутствует со значением по умолчанию.

Если атрибут был описан в DTD с #IMPIED и он отсутствует в документе, то в модели ему не соответствует никакой attribute node.

Атрибут node имеют полное имя, которое определяется аналогично element nodes. Строковое значение attribute node – нормализованное значение соответствующего атрибута. Для атрибутов, вводящих в префиксы для идентификатора пространств имен, attribute node не создается.

NAMESPACE NODES

Каждый element node имеет связанное с ним множество namespace nodes. Каждый элемент этого множества соответствует префиксу, входящему в область видимости элемента. Подобно attribute node для namespace nodes родителем является element node, к которому они относятся.

Element node будет иметь соответствующий namespace node в каждом из следующих случаев:

1. для каждого атрибута элемента, указанного с префиксом xmlns:

2. для каждого атрибута непосредственного или косвенного предка, указанного с префиксом xmlns:, если в данном элементе не было сделано переопределение

3. для атрибута xmlns, указанного на каком-нибудь ближайшем предке, в случае, если его значение не было указано как “ ”.

Полное имя namespace node содержит null в качестве идентификатора пространства имен и префикс пространства имен как локальное имя. Строковые значения namespace node – идентификатор пространства имен, соответствующий префиксу.

PROCESSING-INSTRUCTION NODES

Они создаются для каждой инструкции обработки в документе.

Полное имя содержит null в качестве идентификатора пространства имен и имя инструкции обработки в качестве локального имени. Строковое значение инструкции обработки содержит данные, указанные после имени и до «?>». XML-объявление не является инструкцией обработки.

COMMENT NODES

Они создаются для каждого элемента комментария, не имеет полного имени. Строковое значение – текст комментария в <!-- -->.

TEXT NODES

Текстовые узлы создаются таким образом, чтобы содержать как можно больше текста, идущего последовательно в документе. Таким образом, у текстового узла не может быть ближайшего сестринского узла такого же типа. Не имеют полного имени. Строковое значение – текст, которому соответствует text node.

ЯЗЫК РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ (ДЛЯ ФАСЕТОВ ТИПА PATTERN)

Регулярные выражения состоят из нескольких ветвей (branches) – альтернатив, разделяемых символом |; ветвь может быть и единственной, ветвь|ветвь|…

Ветвь представляет собой 0 или более частей (pieces), каждая из которых содержит, так называемый, атом (atom) и необязательное указание количества его повторений (quantifier), которые задаются одним из трех символом:

1. ? – атом может отсутствовать;

2. + - атом может повторяться один или более раз;

3. * - атом может повторяться 0 или более раз

Количество повторений может быть указано и более точно в виде {число} – указывается точное количество повторений; {число,} – минимальное количество повторений; {число, число} – минимальное количество и максимальное количество повторений.

Атом представляет собой либо символ, либо класс символом, либо регулярное выражение в (). При указании конкретного символа нельзя использовать символы, имеющие специальные значения в языке регулярных выражений: «.», «\», «?», «*», «+», «{», «}», «(», «)», «[», «]». Для их использования в качестве обычных символом необходимо использовать esc последовательности вида: \специальный_символ. Например, \. или \{.

Класс символов задает набор символов и может быть либо предопределенными, либо задан выражением в []: [выражение].

Выражение, задающее класс символов, может быть задано в виде множества допустимых или недопустимых символом, либо путем вычитания допустимых или недопустимых символов.

Множество допустимых символов задается в виде диапазонов: начальный_символ – конечный_символ диапазона. Например, A-Z, A-Za-z

Множество недопустимых символов задается по тем же правилам, но в начале выражения ставится символ ^. Например, ^A-Z.

Вычитание задается в виде: множество_допустимых_символов – [выражение], множество_недопустимых_символов – [выражение].

Предопределенный классы могут задавать множество из одного символа, могут задавать категорию символов или предопределенное множество символом.

Классы, задающие множество из 1-го символа (Single Character Espace): \n – newline (#xA) – перевод строки; \r – return (#xD) – возврат коретки; \t – tab (#x9) – табуляция; \\ - эквивалентно \; \| - эквивалентно |; \. – эквивалентно.; \^ - эквивалентно^; \- - эквивалентно -; \? – эквивалентно ?; * - эквивалентно *; \+ - эквивалентно+; \{ - эквивалентно {; \} – эквивалентно }; \(- эквивалентно (; \) – эквивалентно); \[- эквивалентно [; \] – эквивалентно].

Категории символов задаются в виде: \p{символ_категории}.

Множество символов, образующих комплиментарную категорию, задается в виде: \P{символ)категории}.

Символы категорий. L – любые буквы. Lu – заглавные буквы. Ll – строчные буквы. N – цифры. Nd – десятичные цифры. Nl – цифры, задаваемые буквами. P – знаки пунктуации. Ps – открывающие символы пунктуации. Pe – закрывающие символы пунктуации. Z – любые разделители. Zs – пробелы. Zl – символы новой строки. S – специфические символы. Sm – математические символы. S m – подкласс валют.

Категория может быть задана не символом категории, а именем блока символов UNICODE. Блоки, соответствуют тем или иным языкам. Категория, заданная блоком обозначается: \p{|simя_блока}

Комплиментарная категория в виде: \P{|sимя_блока}. Например, \p{|sBasicLatin}, \P{|sCyrillic}.

Классы, задающие множество символов: . - эквивалентно [^\n\r]. \s – эквивалентно [#x20\t\n\r] – пробельный материал. \S – эквивалентно [^\s] – непробельные символы. \i – символы допустимые для XML имен (с которых XML имена могут начинаться). \I – эквивалентно [^\i] – символы недопустимые для XML имен. \c – допустимые для XML имен. \C – эквивалентно [^\c] – недопустимые для XML имен. \d – эквивалентно \p{Nd} – для десятичных цифр. \D - эквивалентно [^\d] – для недесятичных цифр. \w – символы, допустимые для слов (то есть исключая разделители, символы пунктуации, специальные символы). \W – эквивалентно [^\w] – недопустимые для слов символы.

Лекция 4. Язык XSLT

ЯЗЫК XSLT

XSL Transformation. XSL – Extensible Stylesheet Language. XSLT – язык для описания процедуры преобразования XML-документов в документы другого вида: XML, HTML или текстовые.

Описание выполняется с использованием технологии XML, то есть XSLT, также как и язык схем, является XML-языком. Идентификатор пространства имен для этого языка: `http://www.w3.org/1999/XSL/Transform` (принято использовать префикс `xsl:`).

Корневым элементом XSLT документа является элемент `stylesheet`, внутри которого можно использовать следующие дочерние элементы (top-level element): `import`, `include`, `strip-space`, `preserv-space`, `output`, `key`, `decimal-format`, `namespace-alias`, `attribute-set`, `variable`, `param`, `template`. Эти элементы в теле `stylesheet` могут располагаться в любом порядке. Кроме того, он может содержать элементы из другого пространства имен (не XSLT) при условии, что полное имя этих элементов содержит непустой идентификатор пространства имен.

Элемент `stylesheet` имеет обязательный атрибут `version`, для которого определено значение `1.0`.

Элементы XSLT-документов, не относящиеся к языку XSLT, используются для формирования структуры выходного XML- или HTML-документа, а также внедрения документации.

Элемент `include` используется для блочного конструирования XSLT-документов путем включения одного XSLT-документа в состав другого. URI включаемого документа задается атрибутом `href`. Включение работает на уровне структуры XML-документа, содержащего XSLT-описание. При включении в тело включающего документа помещаются дочерние элементы; они замещают элемент `include` и не влияют на порядок обработки правил преобразования XSLT. Запрещается прямо или косвенно включать самого себя.

Элемент `import` реализует механизм импорта XSLT-документов, который отличается от механизма включения тем, что включенные правила преобразования имеют меньший приоритет, чем правила, содержащие в импортирующем документе. URI импортируемого документа указывается атрибутом `href`. Элементы `import` должны идти первыми.

Технология XSLT использует модель документа и язык выражений, определенный спецификацией XPath. При этом базовая библиотека функций языка XPath расширена некоторым набором функций, специфичных для XSLT.

Элемент `output` позволяет указать, в каком формате будет осуществляться вывод результата XSLT-преобразования.

XSLT-преобразование определяется как преобразование исходного XML-документа в другой XML-документ. Так как XSLT оперирует древовидной моделью документа, то исходный XML-документ часто называют Source XML Tree, а результат преобразования – Result XML Tree. При этом результат преобразования может быть выведен и в не XML виде.

Метод вывода задается атрибутом `method` элемента `output`. В XSLT существует три метода вывода: `xml`, `html`, `text`, также можно указать другие значения, не определенные спецификацией. В последнем случае значение атрибута `method` специфично для XML-процессора.

При отсутствии элемента `output` метод вывода определяется следующим образом: если результат преобразования имеет единственный дочерний элемент корневого узла, его локальное имя `html`, а идентификатор пространства имен – `null`, то метод вывод – `html`, иначе – `xml`.

Остальные атрибуты элемента `output` задают параметры вывода:

1. `version` – указывает версию для метода вывода: это актуально для методов вывода `html` и `xml`. Значения по умолчанию: `html` – `4.0`, `xml` – `1.0`.

2. `encoding` – указывает имя кодировки, в которой осуществляется вывод. Обязательная поддержка: UTF8, UTF16.

3. `media-type` указывает MIME-тип результата вывода: для XML – `text/html`; для `html` – `text/html`; для `text` – `text/plain`.

4. `doctype-public`, `doctype-system` – указывает соответственно публичный и системный идентификатор для DTD, внедряемый в результат вывода (актуальный для методов вывода `html` и `xml`).

5. `omit-xml-declaration` – позволяет указать включать или нет объявление XML в результат вывода для метода `xml`: значение по умолчанию – `no`, если `yes`, то объявление XML не включается.

6. `cdata-section-elements` – актуален для метода `xml`, позволяет указать список имен элементов, чьи текстовые дочерние узлы должны быть выведены в секции CDATA.

ОПИСАНИЯ ШАБЛОНОВ

Описания шаблонов выполняется элементами `template`. Узел или множество узлов, к которым применяется шаблон, указывается атрибутом `match`, значение которого – XPath выражение. Шаблону может быть назначено имя атрибутом `name`, что позволяет вызывать его из других шаблонов. В теле шаблона содержатся инструкции.

Основные инструкции оформляются следующими элементами.

`apply-templates` позволяет применить шаблоны к дочерним элементам тех элементов, к которым применим данный шаблон. Ограничить множество дочерних узлов можно атрибутом `select`.

Существует группа инструкций для создания элементов, атрибутов и текстовых узлов результирующего дерева.

Элементы создаются инструкцией `element`. Атрибут `name` (обязательный) – квалифицированное имя, `namespace` – идентификатор пространства имен, `use-attribute-sets` – список имен групп атрибутов.

Атрибуты создаются инструкцией `attribute`. `name` – обязательный атрибут, `namespace` – идентификатор пространства имен.

Текстовые узлы создаются инструкцией `text`. Отключение обработки esc-последовательностей выполняется атрибутом `disable-output-escaping`. Значение по умолчанию – `no`. Если указано `yes`, то обработка esc-последовательности. Например, «<» не выполняется, то есть для данного примера в результате преобразования будет помещен текстовый узел с содержимым не «<», а с «<».

В теле инструкций `element` и `attribute` указывается шаблон, а теле инструкции `text` – содержимое создаваемого текстового узла.

Инструкции `element` и `attribute` используются для создания вычисляемых узлов, то есть узлов, имена которых могут определяться выражением.

Инструкции обработки создаются инструкцией `processing-instruction`. Атрибут `name` – имя, атрибут `comment` – комментарии.

Именованные группы атрибутов, которые можно использовать при создании элементов, создаются с помощью `top-level-element`, `attribute-set` (обязательный элемент), `name` – имя. В теле указывается инструкция `attribute`.

Генерировать текст выходного документа, обращаясь к элементам и атрибутам входного и используя переменные, можно с помощью инструкции `value-of`. Атрибут `select` (обязательный) указывает XPath-выражение, результат вычисления которого преобразуется в строку по правилам функции `string` и на ее основе создается новый текстовый узел. Атрибут `disable-output-escaping` позволяет отключить обработку esc-последовательностей.

Инструкция `number` позволяет форматировать числа, которые будут включены в выходной документ. Атрибут `value` задает XPath-выражение, результат вычисления которого преобразуется в число по правилам функции `number`, которое затем округляется до целого и преобразуется в строку на основе остальных атрибутов функции `number`. Результат преобразования приводит к созданию нового текстового узла.
Если атрибут `value` не указан, в качестве исходного значения используется позиция текущего узла. При этом учитываются следующие атрибуты: `level` – `single`, `multiple`, `any`; `from` – указывает с какого узла начинается нумерация; `count` – какие узлы будут пронумерованы; `format` – формат преобразования числа в строку, значение по умолчанию – 1; значения этого атрибута расширяют допустимые значения атрибута `type` элемента OL в HTML.

Существует группа инструкций для управления генерацией вывода, такие, как циклы и ветвление.

Для описания циклов используется `for-each`. Атрибут `select` указывает множество узлов для циклической обработки в виде XPath-выражения.

Для ветвления используется пара инструкций:

1. `if` – простое ветвление. Условие, при соблюдении которого производится обработка тела, задается атрибутом `test` в виде XPath-выражения, результат вычисления которого преобразуется к логическому типу по правилам функции `boolean`.

2. `choose` – множественное ветвление. Альтернативы в ее теле описываются инструкциями `when`, условия для которого задаются атрибутами `test`, альтернатива по умолчанию описывается инструкцией `otherwise`.

Инструкция `sort` используется для сортировки. Ее можно использовать как дочерний элемент в теле инструкции `apply-templates` и `for-each`. Выполняет сортировку узлов, к которым применяются инструкции, совместно с которыми она используется. Атрибут `order` указывает порядок сортировки: `ascending` (по возрастанию), `descending` (по убыванию). Атрибут `data-type` указывает тип данных: `text`, `number`, можно указать и другие значения в виде квалифицированного имени, которое поддерживается данной реализацией XSLT-процессора. Атрибут `case-order` используется, если `data-type="text"`, и указывает, какие символы имеют первоочередной порядок: a или A. Значение по умолчанию определяется атрибутом `lang`, в котором задается язык сортируемого текста, если не указать, то его значение определяется из системных настроек.
Атрибут `select` задает выражения, которые вычисляются для каждого обрабатываемого узла, результат вычисления преобразуется в строку по правилам функции `string` и полученное значение используется как ключ сортировки. По умолчанию `select="."`, то есть ключ сортировки – строковое значение узла.

ПЕРЕМЕННЫЕ И ПАРАМЕТРЫ

Переменные и параметры позволяют связать имя с некоторым значением и затем использовать его с выражением. Для этой же цели существуют и параметры. Различие в том, что значение, связанное с параметром, это значение по умолчанию для параметра, используемого при вызове шаблона.

Переменные могут использоваться как элементы верхнего уровня и как инструкции и описываются элементом `variable`.

Параметры описываются элементами `param`, имя параметра задается атрибутом `name`, значение – атрибутом `select` в виде XPath-выражения.

Вызов шаблона, при котором можно использовать параметры, описывается инструкцией `call-template`. Имя шаблона – атрибут `name` (обязательный). Параметры указываются инструкцией `with-param` в теле инструкции `call-template`. Имя параметра задается атрибутом `name` (обязательный), а значение – атрибутом `select` (XPath-выражение). Параметры в выражениях можно использовать в виде `{$имя_параметра}`.

Пример шаблона:

```
<xsl:template name="numbered-block">
   <xsl:param name="format">1.</xsl:param>
   <fo:block>
      <xsl:number format="{$format}"/>
      <xsl:apply-templates/>
   </fo:block>
</xsl:template>
```

Вызов шаблона:

```
<xsl:template match="0|//0|/|i">
   <xsl:call-template name="numbered-block">
      <xsl:with-param name="format">a.</xsl:with-param>
   </xsl:call-tempate>
</xsl:template>
```

ФУНКЦИИ

`system-property` – возвращает значение системного свойства, имя которого указывается в виде квалифицированного имени в качестве параметра. Имена системных свойств: `xsl:version` – возвращает версию; `xsl:vendor` – возвращает название компании – разработчика XSLT-процессора; `xsl:vendor-url` – возвращает ссылку на ее сайт.

`generate-id` – генерирует строку, которая уникальным образом идентифицирует узел, который в множестве узлов, переданном как параметр, идет первым в документе.

`format-number` – преобразует в строку число, указанное первым параметром, в соответствии с форматом, указанным 2-ым параметром, и с использованием именованного формата (3-параметр), который необязателен. Если же он все же указан, он должен соответствовать именованному формату, описанному элементом `decimal-format`, который является элементом верхнего уровня. Формат, указанный 2-м аргументом, описывается в том же синтаксисе, который используется классом `java.text.DecimalFormat`. Имя указывается атрибутом `name`. Атрибут `decimal-separator` указывает символ десятичного разделителя (по умолчанию `.`), `grouping-separator` – символ разделителя групп (`,`), `zero-digit` – символ нуля (`0`), `infinity` – для представления бесконечности (`Infinity`), `NaN` – not a number (`NaN`), `minus-sign` – для представления знака «минус» (`-`). Влияют на интерпретацию форматной строки: `digit` (`#`) – указывает символ, который обозначает цифру; `pattern-separator` (`;`) – указывает символ, который обозначает символ-разделитель для «+» и «-» частей формата строки. Эти атрибуты соответствуют `get`/`set`-методам класса `DecimalFormatSymbols`.

ПРИМЕРЫ

Пример 1. Обработка XML-документа, содержащего документ. Дано DTD-описание документа:

```
<!ELEMENT doc (title, chapter*)>
<!ELEMENT chapter (title, (para|note)*, section*)>
<!ELEMENT section (title, (para|note)*)>
<!ELEMENT title (#PCDATA|emph)*>
<!ELEMENT para (#PCDATA|emph)*>
<!ELEMENT note (#PCDATA|emph)*>
<!ELEMENT emph (#PCDATA|emph)*>
```

XSLT-документ:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/TR/xhtm11/strict">
   <xsl:output method="xml" encoding="ISO-8859-1"/>
   <xsl:template match="doc">
      <html>
         <head>
```

```
            <title>
               <xsl:value-of select="title"/>
            </title>
         </head>
         <body>
            <xsl:apply-templates/>
         </body>
      </html>
   </xsl:template>
   <xsl:template match="doc/title">
      <h1>
         <xsl:apply-templates/>
      </h1>
   </xsl:template>
   <xsl:template match="chapter/title">
      <h2>
         <xsl:apply-templates/>
      </h2>
   </xsl:template>
   <xsl:template match="section/title">
      <h3>
         <xsl:apply-templates/>
   </xsl:templates>
   <xsl:template match="para">
      <p>
         <xsl:apply-templates/>
      </p>
   </xsl:template>
   <xsl:template match="note">
      <p class="note">
         <b>NOTE:</b>
         <xsl:apply-templates/>
      </p>
   </xsl:template>
   <xsl:template match="emph">
      <em>
         <xsl:apply-templates/>
      </em>
   </xsl:template>
</xsl:stylesheet>
```

Пример входного документа XML:

```
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
   <title>Document Title</title>
   <chapter>
      <title>Chapter Title</title>
      <section>
         <title>Section Title</title>
         <para>This is a text.</para>
         <note>This is a note.</note>
      </section>
      <section>
         <title>Another Section</title>
         <para>This is <em>another</em>text.</para>
         <note>This is another note</note>
      </section>
   </chapter>
</doc>
```

Результат преобразования:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns="http://www.w3.org/TR/xhtml11/strict">
   <head>
      <title>Document Title</title>
   </head>
   <body>
      <h1>DocumentTitle</h1>
      <h2>ChapterTitle</h2>
      <h3>Section Title</h3>
      <p>This is a text.</p>
      <p class="note">
         <b>NOTE:</b>This is a note.
      </p>
      <h3>Another Section Title</h3>
      <p>This is<em>another</em>text.</p>
      <p class="note">
         <b>NOTE:</b>This is another note.
      </p>
   </body>
</html>
```

Пример 2. Обработка XML-документа, содержащего данные. Пример XML-документа:

```
<sales>
   <division id=”North”>
      <revenue>10</revenue>
      <growth>9</growth>
      <bonus>7</bonus>
   </division>
   <division id=”South”>
      <revenue>4</revenue>
      <growth>3</growth>
      <bonus>4</bonus>
   </division>
   <division id=”West”>
      <revenue>6</revenue>
      <growth>-1.5</growth>
      <bonus>2</bonus>
   </division>
</sales>
```

XSLT-документ для преобразования документа в HTML (используется упрощенный синтаксис):

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <head>
      <title>Sales Results By Division</title>
   </head>
   <body>
      <table border="1">
         <tr>
            <th>Division</th>
            <th>Revenue</th>
            <th>Growth</th>
            <th>Bonus</th>
         </tr>
         <xsl:for-each select="sales/division">
           <xsl:sort select="revenue" data-type="number" order="descending"/>
            <tr>
               <td>
                  <em><xsl:value-of select="@id"/></em>
```

```
                </td>
                <td>
                   <xsl:value-of select="revenue"/>
                </td>
                <td>
                   <xsl:if test="growth &lt;0">
                      <xsl:attribute name="style">
                         <xsl:text>color:red</xsl:text>
                      </xsl:attribute>
                   </xsl:if>
                   <xsl:value-of select="growth"/>
                </td>
                <td>
                   <xsl:value-of select="bonus"/>
                </td>
             </tr>
          </xsl:for-each>
       </table>
    </body>
</html>
```

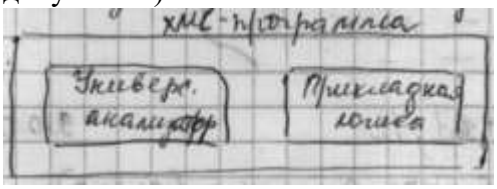
Результат преобразования:

```
<html>
   <head>
     <meta http-equip="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Sales Results By Division</title>
   <head>
   <body>
      <table>
         <tr>
            <th>Division</th>
            <th>Revenue</th>
            <th>Growth</th>
            <th>Bonus</th>
         </tr>
         <tr>
            <th><em>North</em></td>
            <th>10</td>
            <th>9</td>
            <th>7</td>
         </tr>
         <tr>
            <th><em>West</em></td>
            <th>6</td>
            <td style="color:red">-1.5</td>
            <th>2</td>
         </tr>
         <tr>
            <th><em>South</em></td>
            <th>4</td>
            <th>3</td>
            <th>4</td>
         </tr>
      </table>
   </body>
<html>
```

Лекция 5. JAXP

JAVA API FOR XML PROCESSING (JAXP)

Обобщенная структура XML-приложения (приложения, обрабатывающего XML-документы):



JAXP предлагает стандартный API для программного анализа XML-документов, при этом в ходе анализа могут быть выполнены дополнительные функции: например, проверка соответствия документа DTD-описанию или схеме, трансформация XML-документа в соответствии с XSLT-описанием.

В состав JAXP входят три механизма для анализа XML-данных:

1. Представление документа в виде иерархии объектов, которую можно анализировать в соответствии с прикладной задачей, модифицировать и сериализовать в XML-представление (в виде потока символов). Система объектов, в виде которых представляется XML-документ, регламентирована W3C в составе спецификации Document Object Model (DOM), а соответствующий подход к анализу называется DOM-анализом.

2. Событийно-ориентированная обработка документа. XML-анализатор в ходе синтаксического анализа текста XML-документа вызывает для обработки отдельных его частей методы обработчика событий, переданного анализатору при его инициализации прикладной программой. Обработчик событий реализует прикладную логику обработки XML-данных и реализует определенный интерфейс. Соответствующий подход к анализу был разработан независимым сообществом разработчиком и называется SAX-анализом.

3. Поточная обработка документа. Данные XML представляются в виде потока лексем или потока событий, которые прикладная программа может запрашивать одно за другим вместо того, чтобы предоставлять обработчики, которые получали бы события от синтаксического анализатора в определяемом самим анализатором порядке. Соответствующий подход к анализу называется StAX-анализом. Все, что требуется от приложения - это определить тип синтаксически разобранного события, отнести его к соответствующему конкретному типу и использовать соответствующие методы для получения информации, относящейся к событию.

DOM-анализ приводит к интенсивному использованию памяти, так как весь анализируемый документ разбирается, и для него в памяти формируется дерево объектов, при этом прикладной анализ полученного дерева не может быть выполнен до того, как будет полностью построено DOM-дерево, что исключает возможность асинхронной обработки. Достоинством DOM-анализа является возможность модификации XML-данных. При этом оперируют объектами, а не текстовыми строками, что удобно.

SAX-анализ, в отличие от DOM, очень экономно использует память, так как для выполнения событийно-ориентированной обработки не требует загружать весь документ в память. Кроме того, анализ документа может быть выполнен асинхронно: он может считываться блоками из потока ввода, связанного например, с сетевым сокетом, а анализ

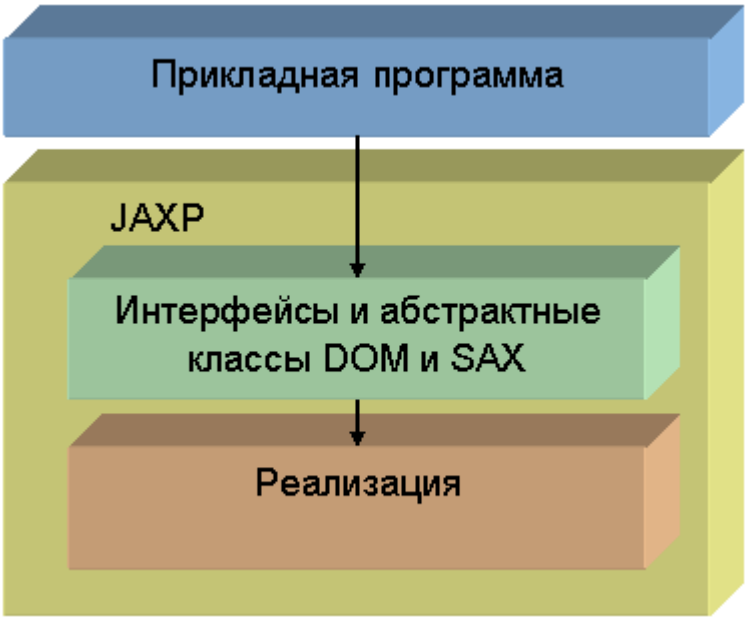
может выполняться по мере поступления блоков, выделения очередных фрагментов XML и обработки соответствующих событий. Перенося фокус с результирующей объектной модели на сам анализирующий поток, приложения получают возможность обрабатывать теоретически бесконечные XML-потоки, поскольку события в своей основе являются временными и не нуждаются в накоплении в памяти. Но такая природа SAX-анализа исключает возможность модификации документа.

При StAX-анализе сохраняются все преимущества SAX-анализа (эффективное использование памяти и асинхронная обработка). Кроме того, поддерживается запись XML-документов и конвейерная обработка. По сравнению с SAX-анализом упрощается разработка прикладной логики обработки, поскольку StAX-анализ не использует обработчики с обратными вызовами, и следовательно приложению не приходится обслуживать эмулированное состояние анализатора.

В составе JAXP можно выделить два слоя:

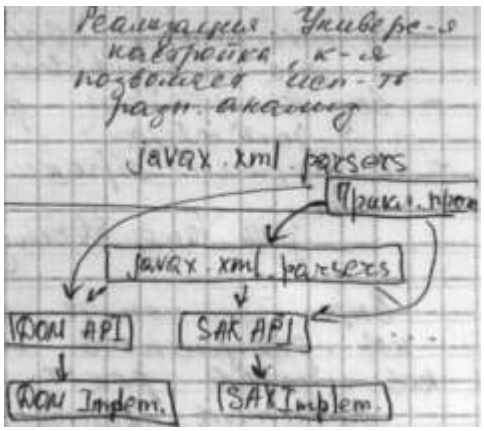
1. API, закрепленный спецификациями Sun JAXP, W3C DOM, SAX, StAX и используемый прикладными программами.

2. Классы, которые отвечают собственно за выполнение синтаксического анализа XML-документа. Они реализуют абстрактные классы из состава JAXP и интерфейсы DOM, SAX и StAX, объекты которых используются для представления информации о любых элементах, извлеченных из XML-документа.



Верхний слой предназначен для использования разработчиком прикладных программ, закреплен в спецификации и относительно стабилен.

Второй слой обеспечивается реализацией платформы Java 2 SE (например, у Sun он свой собственный, у IBM – тоже свой), но он может быть заменен по усмотрению разработчика прикладных программ при запуске программы с помощью системных свойств, определенных спецификацией JAXP. При этом достаточно использовать библиотеки сторонних разработчиков, реализующих JAXP API (например, Apache DOM и SAX анализа).


Реализация. Универс-я
настройка к-я
позволяет исп-ть
разн. анализ
javax.xml.parsers
javax.xml.parsers
DOM API
SAX API
DOM Implem.
SAX Implem.

Лекция 6. DOM-анализ

DOM-АНАЛИЗ

Базовые классы для DOM-анализатора расположены в пакете `javax.xml.parsers`. Для создания объекта парсера используется фабрика, которая представляется объектом абстрактного класса `DocumentBuilderFactory`, а сам парсер – объектом абстрактного класса `DocumentBuilder`.

Объект фабрики создается следующим образом:
```
DocumentBuilderFactory factory = DocumentBulderFactory.newInstance();
```

Метод `DocumentBulderFactory.newInstance()` анализирует конфигурацию, которая указывает, объект какого неабстрактного подкласса необходимо создать, и создает его. При невозможности сделать это, метод выбрасывает `FactoryConfigurationError`.

Для объекта фабрики можно установить определенные свойства, которые будут влиять на работу порождаемых ею объектов парсера. Эти свойства будут рассмотрены далее.

Сам объект парсера создается методом `DocumentBuilderFactory.newDocumentBuilder()`, который может выбросить исключение `ParseConfigurationException`:
```
DocumentBuilder builder = factory.newDocumentBuilder();
```

Объект-парсер далее может использоваться для выполнения DOM-анализа XML-документа. Для этого используется одна из перегруженных версий метода `parse()`, которая в качестве источника XML-данных может использовать: 1) любой поток ввода (`java.io.InputStream`); 2) файл (`java.io.File`); 3) произвольный URI, заданный в виде строки; 4) входной источник (`org.xml.sax.InputSource`). Входной источник предназначен для хранения системного и/или публичного идентификатора XML-сущности (частным случаем которой является XML-документ), а также байтового (`java.io.InputStream`) или символьного (`java.io.Reader`) потока ввода. В любом случае результат анализа представляется в виде объекта типа `org.w3c.dom.Document` – это базовый интерфейс объектной модели документа консорциума W3C.

Кроме того, объекты-парсер могут использоваться как фабрики документов. Для создания документа предназначен метод `newDocument()`, который возвращает пустой документ в виде объекта типа `Document`.

```
Document sourceDocument = builder.parse ("file:///home/John/sourse.xml");
Document destinationDocument = builder.newDocument();
```

После получения объекта типа `Document` можно выполнять далее прикладной анализ XML-данных, используя созданную парсером древовидную систему объектов. Работа с этими объектами ведется исключительно только через реализованные их классами интерфейсы, специфицированные моделью W3C DOM. Эти интерфейсы сгруппированы в пакете `org.w3c.dom`.

СВОЙСТВА ФАБРИКИ DOCUMENTBUILDERFACTORY

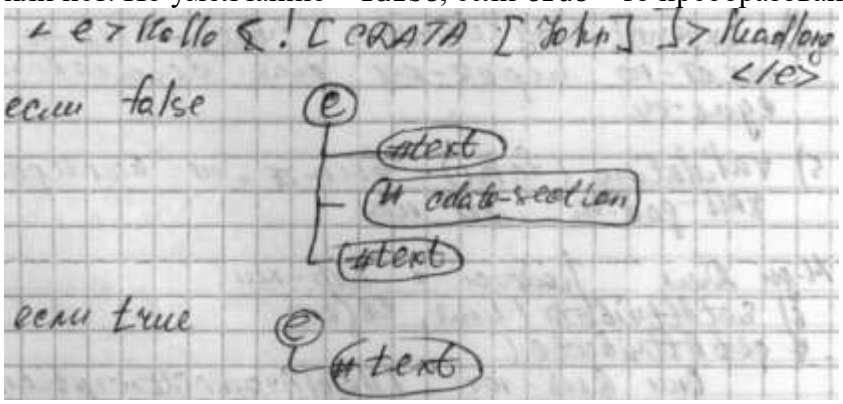
1. Для работы со свойствами фабрики используется подход, принятый в технологии JavaBeans, то есть через get/set-методы. Так как все свойства фабрики – логического типа, то get-метод использует в своем имени префикс is-.

1. `namespaceAware` – указывает парсеру учитывать префиксы имен элементов и атрибутов, связанные с теми или иными идентификаторами пространств имен. Значение по умолчанию – `false`.

Если свойство установлено в `true`, то по результатам анализа для квалифицированных имен можно получать отдельно префиксы и локальные имена. Иначе – разделение квалифицированных имен на эти части не производится, и получить их можно только целиком.

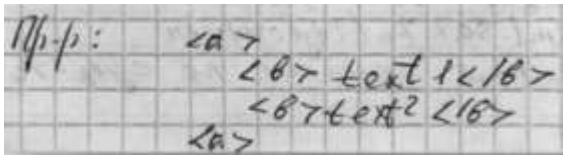
2. `ignoringComments` – контролирует, будет ли парсер включать объекты комментариев в DOM-дерево. По умолчанию – `false`, если `true` – то комментарии игнорируются.

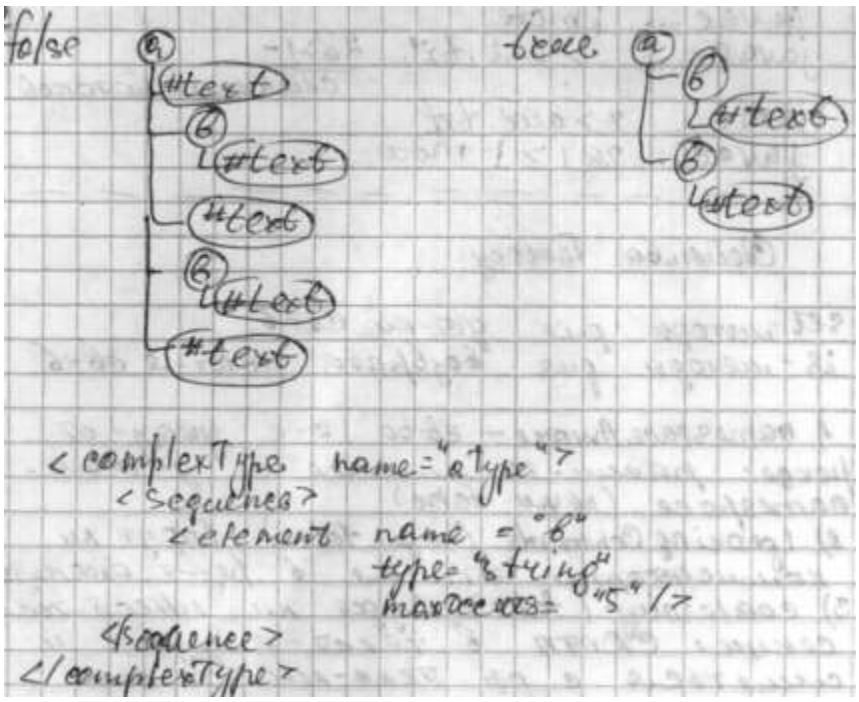
3. `coalescing` – указывает, будет ли выполняться трансформация секций CDATA в текстовые узлы (text nodes) и их слияние со смежными (если они есть) текстовыми узлами или нет. По умолчанию – `false`, если `true` – то преобразование выполняется.



4. `ignoringElementContentWhitespace` – указывает, будет ли парсер включать в результат анализа пробельный материал, который может быть проигнорирован без нарушения соответствия DTD-описанию или схеме анализируемого документа. К такому материалу относится пробельный материал, который расположен непосредственно в теле документа, модель тела (content model), которая допускает присутствие в нем только других элементов. По умолчанию – `false`, если `true` – «лишний» пробельный материал будет проигнорирован. Установка значения `true` требует, чтобы парсер работал в режиме валидации анализируемого документа.

```
<!ELEMENT a(b,c)>
<a>
   <b>…</b>
   <c>…</c>
</a>
```





5. `expandEntityReferences` – контролирует, будут ли в ходе анализа выполняться подстановки для ссылок на сущности или нет. По умолчанию – `true`, если `false` – подстановки выполняться не будут.

6. `validating` – указывает, будет ли выполняться проверка соответствия XML-документа DTD-описанию или схеме. По умолчанию – `false`, если `true` – проверка будет выполняться (парсер будет работать в режиме валидации).

Фабрика представляет еще два метода для работы с атрибутами – `setAttribute()` и `getAttribute()`. Имена, типы значений и семантика атрибутов специфичны для конкретной реализации анализатора. Указанные методы могут выбросить: `IllegalArgumentException`, если реализация не распознала имя свойства.

СВОЙСТВА ПАРСЕРА

У парсера существует только два свойства для записи:

1. `entityResolver`, типа `org.xml.sax.EntityResolver`, содержит объект, который в ходе анализа XML-документа будет использован для выполнения подстановок для ссылок на сущности. По умолчанию – `null`, означает, что будет использован объект класса, специфичного для реализации.

2. `errorHandler`, типа `org.xml.sax.ErrorHandler`, содержит объект, который будут использован для обработки ошибок, возникающих в ходе анализа и управления дальнейшим поведением парсера. По умолчанию – `null`, означает, что будут использоваться объект класса, специфичного для реализации.

УПРАВЛЕНИЕ РЕЖИМОМ ВАЛИДАЦИИ

Валидация анализируемого документа может выполняться либо по схеме, либо по DTD. При этом в документ может быть внедрена ссылка на DTD (элементом DOCTYPE), ссылка на схему (атрибутом `xsi:schemaLocation` или атрибутом `xsi:noNamespaceSchemaLocation`), либо и то и другое.

Спецификация JAXP определяет два стандартных атрибута, которые влияют на проверку документа в соответствии со схемой. Атрибут с именем `http://java.sun.com/xml/jaxp/properties/schemaLanguage` указывает идентификатор языка схем в виде URI. Спецификация JAXP требует, чтобы реализация поддерживала язык схем W3C, идентифицируемый значением `http://www.w3.org/2001/XMLSchema`. Второй атрибут с именем `http://java.sun.com/xml/jaxp/properties/schemaSource` указывает источник, из которого может быть извлечена схема. Значения могут быть типа `String`, `InputSource`, `InputStream`, `File` или массива объектов любого из перечисленных 4-х типов. Эти атрибуты устанавливаются методом `setAttribute()` для фабрики DOM-парсеров в режиме валидации, а также методом `setProperty()` для экземпляра валидирующего SAX-парсера.

Все это порождает несколько возможных ситуаций, для каждой из которых спецификация определяет поведение парсера (табл. 1).

Признаки:
1 – Есть ли элемент `DOCTYPE`
2 – Установлен ли атрибут фабрики `schemaLanguage`
3 – Установлен ли атрибут фабрики `schemaSource`
4 – Есть ли атрибут корневого элемента `schemaLocation`/`noNamespaceSchemaLocation`

Признаки				Проверка ведется по	Используемый файл
1	2	3	4		
Нет	Нет	Нет	Нет	Ошибка, необходим `DOCTYPE`	N/A
Нет	Нет	Нет	Есть	Ошибка, не указан язык схем	N/A
Есть/нет	Нет	Есть	Есть/нет	Ошибка, указан источник схемы, но не указан язык схем	N/A
Есть/нет	Есть	Нет	Есть	XML-схема	Файл схемы, указанный в XML-документе
Есть/нет	Есть	Есть	Нет	XML-схема	Файл схемы, указанный атрибутом `schemaSource`
Есть/нет	Есть	Есть	Есть	XML-схема	Файл схемы, указанный атрибутом `schemaSource`, если целевое пространство имен схемы соответствует XML-документу, иначе файл схемы, указанный в XML-документе

Есть	Нет	Нет	Есть/нет	DTD	DTD, указанный в XML-документе

Пример DOM-анализа XML-документа с проверкой его соответствия схеме:

```
try {
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   factory.setNamespaceAware(true);
   factory.setValidating(true);
   factory.setAttribute(
      "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
      "http://www.w3.org/2001/XMLSchema"
   );
   factory.setAttribute(
      "http://java.sun.com/xml/jaxp/properties/schemaSource",
      "file:///home/john/order.xsd"
   );
   DocumentBuilder builder = factory.newDocumentBuilder();
   Document document = builder.parse("file:///home/john/order.xml");
   // прикладной анализ
   … … …
   } catch (Exception e) {
      e.printStackTrace();
   }
```

DOM PLUGABILITY

Этот механизм позволяет заменить используемую программой реализацию DOM без перекомпиляции. Механизм основан на том, что метод `newInstance()` класса `DocumentBuilderFactory` выполняет поиск неабстрактного подкласса этого класса (с целью создания объекта и возврата его клиенту) в следующей последовательности:

1. проверяется наличие системного свойства с именем `javax.xml.parsers.DocumentBuilderFactory`, если оно установлено, то его значение используется как имя подкласса, объект которого фактически создается.

2. проверяется наличие файла `jaxp.properties` в подкаталоге `lib` каталога установки JRE, содержащего указанное выше свойство

3. просматриваются все доступные виртуальной машине во время выполнения приложения JAR-архивы в поисках файла `META-INF/Services/javax.xml.parsers.DocumentBuilderFactory`, содержащего имя подкласса, объекты которого фактически создаются

4. используется подкласс, специфичный для данной реализации виртуальной машины.

Если неабстрактный подкласс класса `DocumentBuilderFactory` не найден методом `newInstance()`, выбрасывается исключение `FactoryConfigurationError`.

Полученный экземпляр фабрики может быть сконфигурирован установкой свойств соответствующими `set`-методами, которые влияют на порождаемые этой фабрикой объекты парсера. Если фабрика не может создать объект парсера в соответствии с установленными свойствами, то при попытке воздать объект парсер выбрасывает исключение `ParseConfigurationException`.

Пример использования DOM Plugability из командной строки:

```
java –classpath(….jar) –Djavax.xml.parsers.DocumentBuilderFactory=
org.apache.xerces.jaxp.DocumentBuilderFactoryImpl имя_класса_программы
```

DOM API

Результат анализа XML-документа DOM-анализатором представляется в виде системы объектов, имеющей древовидную структуру. Базовые типы для этих объектов расположены в пакете `org.w3c.dom`. Узлы дерева совместимы с типом `Node` – это базовый интерфейс для работы со всеми элементами XML-документа.

Точкой входа в результат анализа является интерфейс `Document`. Объект именно этого типа возвращается в результате анализа.

Интерфейс `Document` расширяет интерфейс `Node`, который расширяется также и другими интерфейсами, предназначенными для работы с любыми элементами XML-документа, формирующими дерево.

ИНТЕРФЕЙС NODE

Различают следующие типы узлов дерева (для каждого типа определена соответствующая константа в интерфейсе `Node`):

1. элемент – ELEMENT_NODE
2. атрибут – ATTRIBUTE_NODE
3. текстовый узел – TEXT_NODE
4. секция CDATA – CDATA_SECTION_NODE
5. ссылка на сущность – ENTITY_SECTION_NODE
6. сущность – ENTITY_NODE
7. инструкции обработки – PROCESSING_INSTRUCTION_NODE
8. комментарий – COMMENT_NODE
9. документ – DOCUMENT_NODE
10. объявление типа документа – DOCUMENT_TYPE_NODE
11. фрагмент документа – DOCUMENT_FRAGMENT_NODE
12. нотация – NOTATION_NODE

Поскольку интерфейс `Node` является базовым для работы с любыми типами узлов, рассмотрим его первым. Методы:

1. `getNodeName()` – возвращает имя узла

2. `getPrefix()` – возвращает префикс

3. `getLocalName()` – возвращает локальное имя

4. `getNodeType()` – возвращает тип узла в виде одной из констант

5. `getParentNode()` – возвращает родительский узел

6. `getChilNodes()` – список дочерних узлов в виде объекта типа `NodeList`

7. `getFirstChild()`, `getLastChild()` – позволяет получить первый и последний из дочерних узлов соответственно

8. `hasChildNodes()` – проверяет наличие дочерних узлов

9. `appendChild()` – добавить новый дочерний узел в конец списка

10. `insertBefore()` – вставить новый дочерний узел перед другим указанным дочерним узлом

11. `replaceChild()` – заменить один дочерний узел на другой

12. `removeChild()` – удалить дочерний узел

13. `getPreviousSibling()`, `getNextSibling()` – получит предыдущий и последующий сестринский узел (узлы первого уровня)

14. `getAttributes()` – получить атрибуты узла в идее объектов типа `NamedNodeMap`

15. `hasAttributes()` – проверить, есть ли у узла атрибуты указанного типа (`NamedNodeMap`)

16. `getOwnerDocument()` – получить ссылку на документ

17. `getNamespaceURI()` – получить идентификатор пространства имен, к которому относится данный узел

18. `lookupNamespaceURI()` - получить идентификатор пространства имен для префикса, указанного параметром

19. `lookupPrefix()` – получить префикс, связанный с идентификатором пространства имен, указанного параметром

20. `isDefaultNamespace()` – проверить, используется ли идентификатор пространства имен, указанный параметром по умолчанию

21. `isEqualNode()` – сравнивать данный узел с другим на равенство по содержимому

22. `compareDocumentPosition()` – сравнить позицию данного узла с позицией другого узла в документе. Метод возвращает одну из констант, определенный в интерфейсе `Node`: `DOCUMENT_POSITION_DISCONNECTED`, `DOCUMENT_POSITION_PRECEDING`, `DOCUMENT_POSITION_FOLLOWING`, `DOCUMENT_POSITION_CONTAINS`, `DOCUMENT_POSITION_CONTAINED_BY`, `DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC`

23. `cloneNode()` – копирование узла, параметр `deep` определяет, будут ли рекурсивно копироваться все вложенные узлы. У копии узла нет родительского узла и связанных с оригиналом пользовательских данных.

Замечание. Любой объект, являющийся узлом DOM-дерева, реализует интерфейс `Node`, причем косвенно, так как интерфейс `Node` расширяется более специфичными

интерфейсами, соответствующими конкретным типам узлов. Например, интерфейс `Element` расширяет интерфейс `Node` и соответствует тем узлам дерева, которые представляют XML-элементы, а интерфейс `Attribute` расширяет интерфейс `Node` и соответствует тем узлам дерева, которые представляют XML-атрибуты. Не все методы, присутствующие в интерфейсе `Node`, имеют смысл применительно к различным конкретным типам узлов. Например, метод `getAttributes()` применим для XML-элементов, а для текстовых узлов смысла не имеет. В случае, когда методы интерфейса `Node` не имеют смысла для конкретного типа узла, возвращается `null`.

Спецификация определяет результат, возвращаемый методами `getNodeName()`, `getNodeValue()`, `getAttributes()` для всех типов узлов:

Интерфейс	`getNodeName()`	`getNodeValue()`	`getAttributes()`
`Attr`	То же, что и `Attr.getName()`	То же, что и `Attr.getValue()`	`null`
`CDATA_Section`	`"#cdata-section"`	То же, что и `CharacterData.getData()`, содержимое CDATA	`null`
`Comment`	`"#comment"`	То же, что и `CharacterData.getData()`, содержимое комментария	`null`
`DocumentFragment`	`"#document-fragment"`	`Null`	`null`
`Document`	`"#document"`	`Null`	`null`
`DocumentType`	То же, что и `DocumentType.getName()`	`Null`	`null`
`Element`	То же, что и `Element.getTagName()`	`null`	`NamedNodeMap`
`Entity`	Имя сущности	`null`	`null`
`EntityReference`	Имя сущности, которой соответствует ссылка	`null`	`null`
`Notation`	Имя нотации	`Null`	`null`
`ProcessingInstruction`	То же, что и `ProcessingInstruction.getTarget()`	То же, что и `ProcessingInstruction.getData()`	`null`
`Text`	`"#text"`	содержимое текстового узла	`null`

Взаимное расположение узлов в дереве, возвращаемое в виде константы методом `compareDocumentPosition()`, определяется на основе поиска общего для обоих узлов контейнера. В частности, значение `DOCUMENT_POSITION_DISCONNECTED` означает, что у узлов нет общего контейнера, что имеет место, когда узлы принадлежат разным документам или фрагментам. Значения `DOCUMENT_POSITION_FOLLOWING` и `DOCUMENT_POSITION_PRECEDING` относятся к случаю, когда у сравниваемых узлов, есть общий контейнер, но ни один из сравниваемых узлов не содержит другой. В случае `DOCUMENT_POSITION_CONTAINED_BY` тот из сравниваемых узлов, который содержится в другом, считается следующим за ним (following), а тот который содержит, считается предыдущим (preceding). То же относится и к случаю `DOCUMENT_CONTAINS`.

В интерфейсе `Node` есть пара методов `getTextContent()`, `setTextContent()`, которые позволяют работать с содержимым узла, то есть с его дочерними узлами и их потомками как с текстом.

Спецификация определяет, что возвращает метод `getTextContent()` для каждого типа узла. Если для узла метод `getTextContent()` возвращает `null`, то вызов `setTextContent()` заменяет содержимое узла единственным текстовым узлом, содержащим текст.

Тип узла	Результат `getTextContent()`
`ELEMENT_NODE, ATTRIBUTE_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, DOCUMENT_FRAGMENT_NODE`	Конкатенация результатов `getTextContent()` для каждого дочернего узла, исключая узлы типа `COMMENT_NODE`, `PROCESSING_INSTRUCTION_NODE`; пустая строка – если дочерних узлов нет
`TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE, PROCESSING_INSTRUCTION_NODE`	То же, что и `getNodeValue()`
`DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE`	`Null`

Интерфейс Node содержит методы, которые позволяют связывать с любым узлом DOM-дерева произвольные данные. Фактически, с каждым узлом может быть связано произвольное количество объектов, идентифицированных некоторыми именами-ключами. Объекты затем можно извлекать по ключу. Связывание объекта с узлом дерева выполняется методом `setUserData()`, а извлечение объекта, связанного с узлом, по ключу – методом `getUserData()`.

При связывании объекта с узлом можно указать также объект-обработчик событий, реализующий интерфейс `UserDataHandler`, метод `handle()` которого будут вызываться для обработки некоторых событий, происходящих с этим узлом. Данный метод в качестве параметров принимает код операции, выполняемой с узлом, ключ, пользовательский объект, а также ссылки на оригинальный и вновь созданный узлы.

Код операции идентифицирует событие, для обработки которого и вызван данный метод; в качестве значения параметра передается одна из констант, определенной в этом же интерфейсе:
1. NODE_ADOPTED – над узлом выполняется операция `Document.adoptNode()`
2. NODE_CLONED – над узлом выполняется операция `Node.cloneNode()`
3. NODE_DELETED – узел удаляется
4. NODE_IMPORTED - над узлом выполняется операция `Document.importNode()`
5. NODE_RENAMED - над узлом выполняется операция `Document.renameNode()`

Оригинальный узел – тот, который подвергается клонированию, адаптации, импорту или переименованию. Если узел удаляется, то в качестве оригинального узла передается `null`. Если в результате операции не создается нового узла, то в качестве вновь созданного узла передается `null`.

ИНТЕРФЕЙС ELEMENT

Интерфейс `Element` расширяет интерфейс `Node` и добавляет методы, специфичные для XML-элементов:

1. `getTagName()` – возвращает имя элемента

2. `getAttribute()`, `setAttribute()` – получить или установить значение атрибута в виде строки по имени

3. `removeAttribute()` – удалить атрибут по имени

Для последних 3-х методов существуют аналоги, оперирующие как объектом типа `Attr`: `getAttributeNode()`, `setAttributeNode()`, `removeAttributeNode()`. `hasAttribute()` – проверить, если ли у элемента атрибут с указанным именем. `setIdAttribute()` – позволяет пометить атрибут с указанным именем как идентифицирующий атрибут или снять эту пометку. Результат этого метода влияет на результат метода `Attr.isId()`, вызванного на помеченном атрибуте и метода `Document.getElementById()`. `setIdAttributeNode()` – атрибут указывается объектом типа `Attr`. Существует группа методов для работы с атрибутами с учетом пространства имен, к которым они относятся. Эти методы идентифицируют атрибуты парой *`идентификатор_пространства_имен – локальное_имя`*
`getAttributeNS()`, `setAttributeNS()`, `removeAttributeNS()`, `getAttributeNodeNS()`, `setAttributeNodeNS()`, `hasAttributeNS()`, `setIdAttributeNS()` – их назначение аналогично назначению соответствующих методов без суффиксов NS.

Узлы типа `Element` и `Attr` позволяют узнать их тип: получить его в виде объекта типа `TypeInfo` можно с помощью метода `getSchemaTypeInfo()`. Объект типа `TypeInfo` является тонкой оболочкой над двумя полями: именем типа, которое можно получить методом `getTypeName()` и пространством имен, к которому относится этот тип – `getTypeNamespace()`. Кроме того, он содержит информацию о наследовании, то есть, был ли создан данный тип на базе другого и каким образом. Это обозначается одним из значений:
1. `DERIVATION_RESTRICTION` – ограничение
2. `DERIVATION_EXTENSION` – расширение
3. `DERIVATION_UNION` – объединение
4. `DERIVATION_LIST` – список

Информация, содержащаяся в объекте `TypeInfo`, соответствует той, что присутствует в DTD или XML-схеме, связанной с документом.

ИНТЕРФЕЙС ATTR

Расширяет интерфейс `Node` и добавляет методы для работы с атрибутами XML-документов. Объекты типа `Attr` представляют в DOM-дереве атрибуты элементов. Методы:

1. `getName()` – получить имя атрибута

2. `getValue()` – получить значение атрибута

3. `setValue()` – установить значение атрибута

4. `getSpecified()` – позволяет определить был ли атрибут указан в документе или он был создан при построении DOM-дерева для атрибута по умолчанию

5. `getOwnerElement()` – возвращает объект `Element`, к которому относится данный атрибут

6. `isId()` – является ли этот атрибут идентифицирующим (типа «ID»)

ИНТЕРФЕЙС CHARACTERDATA

Расширяет интерфейс `Node` и является базовым для других интерфейсов, предназначенных для работы с текстовыми данными. Методы:

1. `getData()` – извлечь текстовые данные

2. `setData()` – изменить текстовые данные

3. `getLength()` – получить длину тестовых данных

4. `substringData()` – получить фрагмент текстовых данных, смещение и длина подстроки указывается параметром

5. `appendData()` – добавить в конец строки

6. `insertData()` – встравить текстовые данные, которые вместе со смещением указываются в качестве параметра

7. `deleteData()` – удалить фрагмент, смещение и длина указываются параметром

8. `replaceData()` – заместить один фрагмент текста данных другим; смещение и длина замещаемого фрагмента указывается параметром.

ИНТЕРФЕЙС TEXT

Интерфейс `Text` расширяет интерфейс `CharacterData` и используется для работы с текстовыми узлами DOM-дерева. Текстовые узлы создаются для каждого текстового фрагмента, не представляющего собой разметку, включая значения атрибутов.

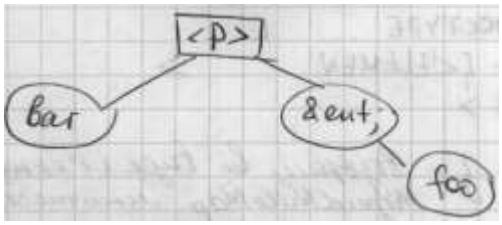
Методы:

1. `splitText()` – разбивает текстовый узел на два, граница разбиения указывается как смещение в виде параметра; возвращает второй (новый) текстовый узел. После разбиения данный узел содержит фрагмент до разбиения (до смещения), а новый – фрагмент данных после смещения. Новый текстовый узел становится ближайшим сестринским узлом данного

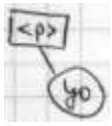
2. `isElementContentWhitespace()` – позволяет определить, представляет ли данный текстовый узел пробельный материал, который может быть проигнорирован без ущерба для структуры и содержания документа (“ignorable whitespace”)

3. `getWholeText()` – возвращает содержимое всех соседних для данного тестовых узлов, соединенных в одну строку в соответствии с их порядком в документе

4. `replaceWholeText()` – позволяет заместить данный текстовый узел и все соседние текстовые узлы новым узлом, содержимое которого указывается в качестве параметра.



`barTextNode.getWholeText()` эквивалентно `"barfoo"`
`barTextNode.replaceWholeText("yo")` эквивалентно



ИНТЕРФЕЙС COMMENT

Данный интерфейс расширяет `CharacterData`. Объекты типа `Comment` представляют в DOM-дереве комментарии. Дополнительных методов не содержит

ИНТЕРФЕЙС CDATASECTION

Данный интерфейс расширяет интерфейс `Text`, объекты типа `CDATASection` представляются в DOM-дереве секции CDATA. Дополнительных методов не содержит.

ИНТЕРФЕЙС DOCUMENTTYPE

Это интерфейс расширяет интерфейс `Node`, объекты типа `DocumentType` – в DOM-дереве – объявления типа документа.

Методы:

1. `getName()` – получить имя типа `Document`, то есть имя корневого элемента, следующего после `<!DOCTYPE…>`

2. `getPublicId()` – возвращает публичный идентификатор

3. `getSystemId()` – возвращает системный идентификатор типа документа

4. `getIntenalSubset()` – возвращает часть DTD, описанную непосредственно в элементе DOCTYPE в виде одной строки; ограничивающие квадратные скобки не включаются

```
<!DOCTYPE … [
   <!ELEMENT … >
```

```
   ]>
```

5. `getEntities()`, `getNotations()` – возвращают в виде объектов типа `NamedNodeMap` множества соответственно общих сущностей (как внутренних, так и внешних) и нотаций, объявленных в DTD; в этих множествах общие сущности представляются объектами типа `Entity`, а нотации – типа `Notation`

ИНТЕРФЕЙС ENTITY

Этот интерфейс расширяет `Node`, объекты типа `Entity` в DOM-дереве представляют сущности, объявленные в DTD (имеются в виду как анализируемые, так и неанализируемые сущности). Если XML-процессор работает в невалидирующем режиме, то сущности, объявленные во внешних DTD-описаниях, могут не учитываться. Узлы типа `Entity` не имеют родительного узла. Для анализируемых сущностей их замещающий текст представляется в виде поддерева, корнем которого является объект типа `Entity`.

Методы:

1. `getPublicId()`, `getSystemId()` – получить идентификатор, ассоциированный с сущностью

2. `getNotationName()` – имя нотации, для неанализируемых сущностей

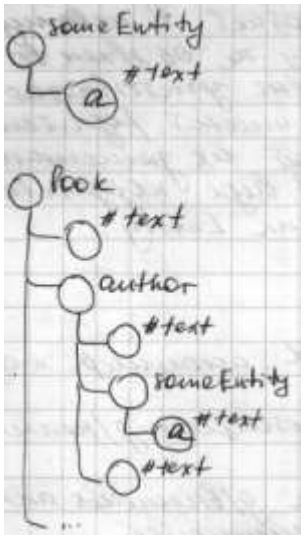
3. `getInputEncoding()` – кодировка для внешних анализируемых сущностей

4. `getXmlEncoding()`, `getXmlVersion()` – возвращает кодировку и версию соответственно, которые указываются в текстовом объявлении внешней анализируемой сущности.

ИНТЕРФЕЙС ENTITYREFERENCE

Этот интерфейс расширяет `Node`; объекты в DOM-дереве – ссылки на сущности (общие). Объекты типа `EntityReference` являются корневыми узлами для поддеревьев, соответствующие замещающему тексту. Эти поддеревья являются корнями поддеревьев, соответствующих объектов типа `Entity`. Узлы, входящие в эти поддеревья и для объектов типа `Entity` и для объектов типа `EntityReference` являются неизменными. Не содержит дополнительных методов.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book [
   <!ELEMENT book (author, title)>
   … … …
   <!ENTITY someEntity, "a">
]>
<book>
   <author>…&someEntity;…</author>
</book>
```



ИНТЕРФЕЙС PROCESSINGINSTRUCTION

Этот интерфейс расширяет `Node`; объекты типа `ProcessingInstruction` – инструкции обработки в DOM-дереве.

Методы:

1. `getTarget()` – получить компонент target

2. `getData()`, `setData()` – получит или изменить компонент data.

```
<?target data?>
```

ИНТЕРФЕЙС DOCUMENT

Этот интерфейс расширяет `Node`; объект типа `Document` является корневым узлом DOM-дерева и является точкой входа в результате DOM-анализа; кроме того, он может быть использован как фабрика для объектов, реализующих прочие DOM интерфейсы.

Фабричные методы: `createElement(v)`, `createAttribute(v)`, `createTextNode(*)`, `createComment(*)`, `createCDATASection(*)`, `createEntityReference(v)`, `createProcessingInstruction()`, `createDocumentFragment()`. Эти методы позволяют создать объекты, реализующие соответственно интерфейсы для программного конструирования DOM-дерева.

Для методов помеченных v параметром указываются имена (элемента/атрибута/сущности).

Для методов * параметров указываются их содержимое.

Для `ProcessingInstruction` – параметром указываются элементы target и data.

Для `createElement()` и `createAttribute()` есть версии, работающие с пространством имен: `createElementNS()`, `createAttributeNS()`.

Прочие методы: `getDoctype()` – получить DTD в виде объекта `DocumentType`. `getDocumentElement()` – получить корневой элемент документа. `getInputEncoding()` – получит кодировку документа. `getXmlEncoding()`, `getXmlVersion()` – получить кодировку и версию, указанную в объявлении XML. `setXmlVersion()` – изменить версию.

Методы поиска элементов:

1. `getElementById()` – получить элементы по значению его идентифицирующего атрибута, указанного как параметр; элемент возвращается в виде объекта типа `Element`.

2. `getElementsByTagName()` – найти элементы по имени; объекты возвращаются в виде объекта типа `NodeList`.

3. `getElementsByTagNameNS()` – аналогичный метод, работает с пространством имен

Методы перемещения/копирования элементов:

1. `adoptNode()` – позволяет переместить узел и связанное с ним поддерево узлов из одного документа в другой, при этом из исходного документа это поддерево удаляется. Узел – корень перемещаемого поддерева указывается параметром метода. Метод возвращает узел – корень поддерева после перемещения.

Операция `adopyNode()` выполняет изменение атрибута `ownerDocument` исходного узла таким образом, чтобы оно указывало на тот документ, для которого был вызван метод `adoptNode()`. Подобным же образом при этом обрабатываются все узлы-потомки и все узлы-атрибуты, если они имеются. Кроме того, узел-корень перемещаемого поддерева удаляется из списка дочерних узлов родителя в оригинальном документе. Операция `adoptNode()` поддерживается не для всех узлов. Для каждого типа узла имеется своя специфика:

1) `ATTRIBUTE_NODE` – атрибут `ownerElement` устанавливается в `null`, флаг `specified` устанавливается в `true`, узлы-потомки атрибута обрабатываются рекурсивно

2) `DOCUMENT_FRAGMENT_NODE` – при выполнении `adoptNode()` для узла типа `DocumentFragment` операция выполняется и для этого узла, и для его потомков.

3) `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `ENTITY_NODE`, `NOTATION_NODE` – для узлов этих типов операция `adoptNode()` не поддерживается

4) `ELEMENT_NODE` – вместе с узлом-элементом переносятся только те из узлов-атрибутов, которые в исходном документе были указаны явно; узлы-потомки обрабатываются рекурсивно

5) `ENTITY_REFERENCE_NODE` – операция `adoptNode()` выполняется только для самого узла этого типа, его потомки не обрабатываются. Если новый документ определяет значение для соответствующей сущности, то для перемещенной ссылки формируется соответствующие поддерево

6) `PROCESSING_INSTRUCTION_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `COMMENT_NODE` – узлы этих типов обрабатываются обычным образом.

2. importNode() – выполняет импорт узла и, возможно, соответствующего ему поддерева без его удаления из исходного документа путем копирования. Импортируемый узел указывается в качестве 1-го параметра, 2-м параметром типа boolean указывается, выполнять импорт поддерева или нет (параметр deep).

При копировании узла создается новый узел с такими же атрибутами nodeName и nodeType, а также атрибутами, относящимися к NS: prefix, localName, namespaceURI. Операция importNode() поддерживается не для всех типов узлов. Для каждого типа узла имеется своя специфика:

1) ATTRIBUTE_NODE – атрибут ownerElement устанавливается в null, атрибут specified устанавливается в true, узлы-потомки атрибута обрабатываются рекурсивно вне зависимости от значения параметра deep

2) DOCUMENT_FRAGMENT_NODE – если deep==true, то узлы-потомки рекурсивно импортируются, иначе создается пустой узел типа DocumentFragment

3) DOCUMENT_NODE, DOCUMENT_TYPE_NODE – узлы этих типов не могут быть импортированы

4) ELEMENT_NODE – при копировании элемента вместе с ним копируются только те из его атрибутов, которые были явно указаны в исходном документе.

5) ENTITY_NODE – узлы этого типа могут быть импортированы, несмотря на то, что узлы типа DocumentType – неизменяемые. При копировании сущности копируются ее атрибуты: publicId, systemId, notationName.

6) ENTITY_REFERENCE_NODE – копируется лишь сам узел этого типа, узлы-потомки не копируются, даже если deep==true.

7) NOTATION_NODE – узлы могут быть импортированы, несмотря на то, что узлы типа DocumentType неизменяемые. При копировании узла копируются его атрибуты: publicId, systemId. Значение deep не оказывает никакого влияния.

8) PROCESSING_INSTRUCTION_NODE – узлы могут быть импортированы; при копировании узла копируются атрибуты target, data. Значение deep не оказывает влияния. Узлы не могут иметь потомков.

9) TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE – при копировании узлов, копируются их атрибуты data, length. Значение deep не оказывает влияния. Узлы этого типа не могут иметь потомков

3. renameNode() позволяет переименовывать узел типа Element или Attr; переименованный узел, а также его новый namespaceURI и квалифицированное имя указывается как параметр.

ИНТЕРФЕЙС DOCUMENTFRAGMENT

Этот интерфейс расширяет интерфейс Node; объекты типа DocumentFragment служат для представления фрагментов документа и выступают в роли тонкой оболочки над, в общем случае, некоторым множеством узлов-вершин поддеревьев.

Фрагмент документа в результате может не быть well-formed документом с точки зрения спецификации XML.

Объекты `DocumentFragment` обрабатываются специфичным образом, когда они используются в операциях вставки узла в документ: при этом вставляется не сам узел `DocumentFragment`, а его дочерние узлы, таким образом, используя `DocumentFragment` можно выполнить вставку множества узлов одной операцией (например, в операциях `Node.appendChild()`, `Node.insertBefore()`).

Схема иерархии наследования основных DOM интерфейсов:

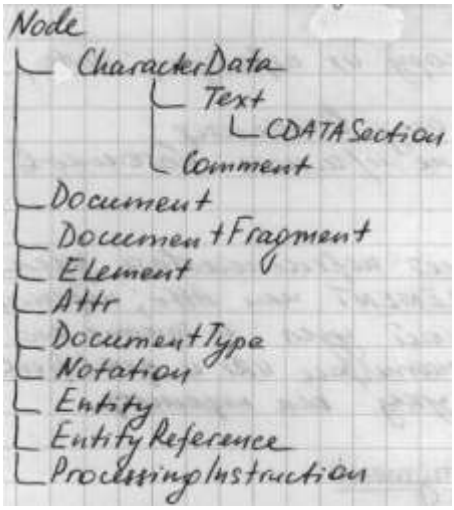
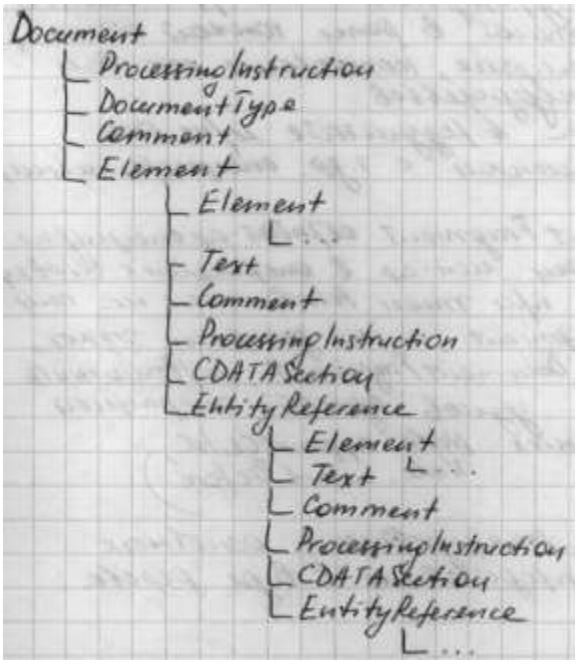
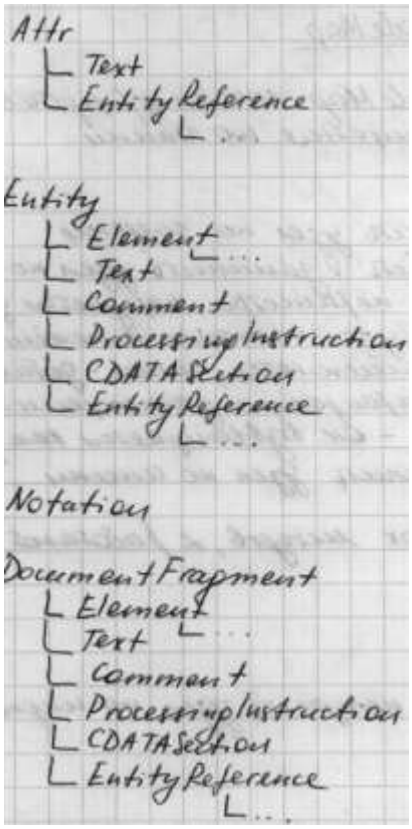


Схема иерархии агрегирования объектов DOM-дерева:





ИНТЕРФЕЙС NODELIST

Объекты типа `NodeList` используются для представления списка узлов; объекты этого типа возвращаются как результаты некоторых методов интерфейсов `Node` и `Document`. Методы:

1. `getLength()` позволяет получить длину списка узлов

2. `item()` позволяет получить узел по его индексу; индеек начинается с нуля

ИНТЕРФЕЙС NAMEDNODEMAP

Объекты типа `NamedNodeMap` используются для представления множеств узлов, к которым можно обращаться по имени. Методы:

1. `getNamedItem()` получить узел по имени

2. `setNamedItem()` добавить или заменить узел по имени. В качестве параметра передается узел; если узел с таким именем уже существует – он заменяется, иначе – добавляется; если произошло замещение узла – он возвращается как результат

3. `removeNamedItem()` удалить узел по имени

4. `item()` – позволяет получить узел по индексу

Версии рассмотренных методов, которые работают с NS: `getNamedItemNS()`, `setNamedItemNS()`, `removeNamedItemNS()`.

РЕАЛИЗАЦИЯ DOM

Спецификация DOM содержит несколько модулей, каждый из которых содержит свое множество интерфейсов и отвечает за свою часть объектной модели документа (реализует свои features). При этом реализация DOM необязательно должна поддерживать все модули. Кроме того, в дальнейшем могут быть добавлены модули, реализующие features, ориентированные, например, на работу с конкретным типом документов, например с SVG документами. Это означает, что прикладным программам необходим механизм, который позволяет на этапе выполнения определить, с какой реализацией DOM они сейчас работают и проверить, поддерживает ли эта реализация необходимые features.

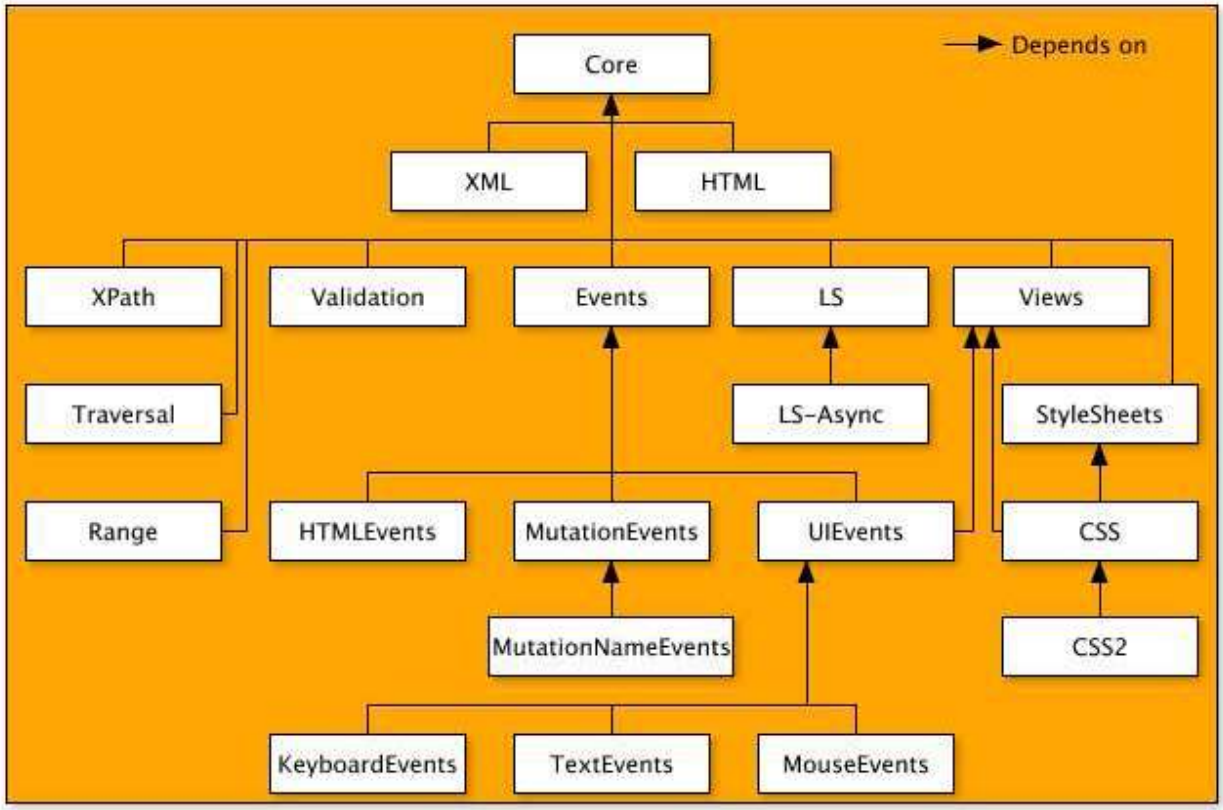
Для этого используется интерфейс `DOMImplementation`. Объекты типа `DOMImplementation` могут быть получены методом `getImplementation()` интерфейса `Document`. Этот интерфейс содержит методы: `hasFeature()` – позволяет проверить, поддерживает ли данная реализация DOM указанную feature, имя и версия которой передается параметром. Если feature поддерживается, то DOM предусматривает два механизма для получения соответствующей реализации:

1. непосредственное преобразование объекта типа `DOMImplementation` к нужному типу.

2. метод `getFeature()` интерфейса `DOMImplementation`. Для получения объекта в виде Object, тип которого можно затем непосредственно привести к нужному типу.

Текущая версия DOM “3.0” (Level 3) содержит следующие модули:

1. Core module, содержит основные интерфейсы, необходимые для работы с любыми документами (HTML, XML и т.д.) – “Core”
2. XML module, содержит интерфейсы, необходимые для работы с XML-документами – “XML”
3. Events module, содержит средства для обработки событий, связанных с манипуляцией документом – “Events”
4. User Interface Events module – “UIEvents”
5. Mouse Events module – “MouseEvents”
6. TextEvents module, обработка событий, связанных с вводом текста – “TextEvents”
7. KeyboardEvents module – “KeyboardEvents”
8. MutationEvents module, обработка событий, связанных с изменением структуры и содержимого документа (вставка и удаление узлов дерева, изменение символьных данных и значения атрибута) – “MutationEvents”
9. MutationNameEvents module, обработка событий, связанных с изменением имени элемента или атрибута – “MutationNameEvents”
10. HTMLEvents module – “HTMLEvents”
11. LoadandSave module – “LS”
12. AsynchronousLoad module = “LS-Async”
13. Validation module, позволяет проводить валидацию в процессе изменения документа – “Validation”
14. XPath module – “XPath”



Если имя feature указано с префиксом «+» в вызове hasFeature – это означает, что в случае true механизм непосредственного приведения типов, возможно, использовать нельзя. Если «+» не был указан, то true возвращает лишь в том случае, если прямое преобразование можно использовать.

Пример:

```
Document doc = docBuilder.parse(…);
DOMImplementation domImpl = doc.getImplementation();
if(domImpl.hasFeatures("LS","3.0")) {
   System.out("LS is not supported");
} else {
   DOMImplementationLS domImplLS = (DOMImplementationLS) domImpl;
   … … …
}
```

Методы, аналогичные рассмотренным, предусмотрены и в интерфейсе `Node`: `isSupported()`, `getFeature()`. В интерфейсе `DOMImplementation` предусмотрена пара фабричных методов, которые отсутствуют в интерфейсе `Document`: `createDocumentType()` – для создания объектов типа `DocumentType`; `createDocument()` – для создания объектов типа `Document`.

КОНФИГУРАЦИЯ ДОКУМЕНТА

В интерфейсе `Document` предусмотрен метод `normalizeDocument()`, которым можно выполнять изменение документа в соответствии с настройками, которые указываются в виде объекта типа `DOMConfiguration`. Связывание этого объекта с документом

выполняется при создании документа, а получить его можно методом `getDOMConfig()` интерфейса `Document`.

Объект `DOMConfiguration` является тонкой оболочкой над совокупностью свойств, называемых параметрами, для работы с которыми предусмотрены методы `setParameter()` и `getParameter()`. Параметры могут быть разных типов. Часть из них обязательно должна поддерживаться любой реализацией DOM. Узнать список параметров, поддерживаемых данной реализацией, можно методом `getParameterNames()`, а узнать возможность установки параметра в то или иное значение можно методом `canSetParameter()`.

При нормализации документа в зависимости от установок может выполняться, например, преобразование секции CDATA в текстовые узлы и их слияние с соседними текстовыми узлами; удаление комментариев, удаление текстовых узлов, содержащих пробельный материал, который может быть проигнорирован; удаление узлов типа `EntityReference` и их замена соответствующими поддеревьями и т. д.

Кроме того, используя параметры `well-formed`, `validate-if-schema` и `validate` можно выполнять проверку соответствия документа требованиям спецификации XML и или описанию его структуры (DTD, схема).

Важно то, что значение параметров `DOMConfiguration` влияет не только на выполнение операции `normalizeDocument()`, но и на работу методов интерфейсов `DOMParser` и `DOMSerializer` из модуля DOM Load and Save.

ОБРАБОТКА ОШИБОК

Обработку ошибок, происходящих в ходе нормализации можно выполнять с помощью объекта, реализующего интерфейс `DOMErrorHandler`, назначив его значением свойства `error-handler` объекта `DOMConfiguration`.

Интерфейс `DOMErrorHandler` содержит только метод `handleError()`. Возникшая ошибка передается как объект типа `DOMError`.

Из объекта `DOMError` можно получить степень тяжести ошибки методом `getSeverity()` в виде одной из констант:
1. `SEVERITY_WARNING` - предупреждение
2. `SEVERITY_ERROR` – ошибка
3. `SEVERITY_FATAL_ERROR` – фатальная ошибка

а также сообщение об ошибке – методом `getMessage()`, связанные с ошибкой исключения (`getRelatedException()`) и данные (`getRelatedData()`). Данные возвращаются в виде объекта, имя типа которого получить методом `getType()`. Метод `getLocation()` позволяет получить объект типа `DOMLocator`, описывающий место документа, в котором возникла ошибка (номер строки, символа, узел документа и т. д.).

При работе с DOM базовым исключение является `DOMException`, которое может выбрасываться любыми методами DOM-интерфейсов. Это исключение – тонкая оболочка над кодом ошибки, которое можно получить методом `getCode()`. Возвращаемое значение соответствует одной из констант, определенной в этом интерфейсе.

Спецификация определяет для каждого метода каждого DOM-интерфейса, может ли он выбрасывать `DOMException` или нет, и если да, то с какими кодами и в каких случаях.

API DOM LOAD & SAVE

API DOM Load and Save предназначен для реализации возможности загрузки XML-документа из источника и сохранения его в некоторый источник.

На уровне Java используются стандартные Java-типы для представления этих источников и работы с ним. Источники бывают бинарными и текстовыми. Для загрузки документа из бинарного источника используется `java.io.InputStream`, из текстового – `java.io.Reader`, `java.io.OutputStream` – для сохранения в бинарный источник, `java.io.Writer` – для сохранения в текстовый источник.

Загрузка и сохранение может выполняться в синхронном и асинхронном режиме.

Для работы с источниками любых типов используются два DOM-интерфейса: `LSInput` представляет источник ввода, а `LSOutput` – источник вывода. Объекты этих типов используются как тонкие оболочки над несколькими (в общем случае) источниками различных типов, из которых фактически будет использован только один – какой, определяется специальным правилом.

Ввод может быть осуществлен из 5-ти видов источников:
1. текстовый (символьный) поток ввода (`characterStream`)
2. байтовый поток ввода (`byteStream`)
3. строка (`stringData`)
4. источник, указанный системным идентификатором (`systemId`)
5. источник, указанный публичным идентификатором (`publicId`)

Эти источники могут быть получены соответствующими `get`-методами интерфейса `LSInput`, а установлены соответствующими `set`-методами. Использован фактически будет тот источник, который не `null`, приоритет соответствует порядку, в котором были перечислены типы источников.

Вывод может быть осуществлен в источник одного из 3-х типов:
1. символьный поток вывода (`characterStream`)
2. байтовый поток вывода (`byteStream`)
3. источник, указанный системным идентификатором (`systemId`)

Кодировка вводимых и выводимых данных в любом случае указывается методом `setEncoding()`.

Для выполнения ввода/вывода предусмотрены интерфейсы:
1. `LSParser`
2. `LSSerializer`

`LSParser` выполняет загрузку и анализ документа из источника, указанного параметром типа `LSInput` метода `parse()`. Результат метода – объект типа `Document`, соответствующий загруженному документу. Загрузку и анализ также можно выполнить, используя URI документа – методом `parseURI()`.

`LSSerializer` – выполняет сохранение документа или любого поддерева, корень которого представлен объектом типа `DOMConfiguration`, который может быть получен методом `getDOMConfig()`. Состав параметров `DOMConfiguration` спецификацией Load and Save расширен несколькими специфическими параметрами.

В процессе загрузки или сохранения можно выполнять фильтрацию узлов документа, с которыми ведется работа, с помощью объектов типа `LSParserFilter` (для `LSParser`) и `LSSerializerFilter` (для `LSSerializer`). В любом случае установка фильтра выполняется методом `setFilter()`.

Фабрикой для объектов типа `LSInput`, `LSOutput`, `LSParser` и `LSSerializer` является объект типа `DOMImplementationLS`, который может быть получен с использованием одного из рассмотренных ранее механизмов (приведением типа объекта `DOMImplementation` или методом `getFeature()` этого объекта – в этом случае в качестве параметра указывается имя модуля “LS” – для работы в синхронном режиме и “LSAsync” – для работы с асинхронном и асинхронном режиме).

При создании объекта типа `LSParser` указывается режим загрузки в виде одной из констант: `MODE_SYNCHRONOUS`, `MODE_ASYNCHRONOUS`. Вторым параметром указывается идентификатор языка схем, в случае, если при загрузке будет выполнятся валидация.

Для языка схем W3C указывается следующий URL: `http://www.w3.org/2001/XMLSchema`. Для DTD: `http://w3.org/TR/REC-xml`.

При загрузке в асинхронном режиме методом `getBusy()` интерфейса `LSParser` можно контролировать, окончилась загрузка или нет.

Спецификация DOM Load & Save вводит специальное исключение, которое может выбрасываться методами рассмотренных интерфейсов – `LSException`. Определены дополнительные коды ошибок.

ВОПРОСЫ, ОСТАВШИЕСЯ ЗА КАДРОМ

При работе с объектом типа `LSParser` в асинхронном режиме методы `parse()` и `parseURI()` возвращают `null`, так как управление возвращается мгновенно, а загрузка документа происходит в отдельном потоке параллельно с клиентским приложением.

Предполагается, что объект, реализующий интерфейс `LSParser`, реализует также и интерфейс `EventTarget`, определенный спецификацией DOM Events. Этот интерфейс позволяет прикрепить к объекту типа `LSParser` обработчики событий, генерируемые этим объектом в процессе загрузки документа. Этих событий два: `progress` – уведомляет о прогрессе в процессе загрузки документа; `load` – уведомляет об окончании загрузки документа.

Обработка этих событий выполняется объектами, реализующими соответствующие интерфейсы прослушивания событий – именно эти объекты прикрепляются к объекту типа `LSParser` для обработки этих событий.

Информация о самих событиях передается объектом типа `LSParser` объекту-обработчику событий в виде объекта, реализующего интерфейс `LSProgressEvent` (для события

progress) или LSLoadEvent (для событий load). Из переданных объектов можно извлечь информацию о соответствующем событии:

1. Интерфейс LSProgressEvent позволяет получить источник (LSInput), из которого происходит загрузка (getInput()), а также позицию, до которой был загружен документ (getPosition()) и общий размер документа (getTotalSize())

2. Интерфейс LSLoadEvent позволяет получить источник (getInput()) и сам загруженный документ (getNewDocument())

Прерывание процесса асинхронной загрузки можно выполнить методом abort() интерфейса LSParser.

В процессе загрузки и анализа документа иногда необходимо загружать также и внешние сущности, ссылки на которые присутствуют в документе. Эти сущности могут храниться в специальных источниках (например, в БД) и требовать для всего извлечения специальной обработки. В этом случае необходимо обеспечить объект, реализующий интерфейс LSResourseResolver, и прикрепить его к объекту типа LSParser, используя параметр с именем «resourse-resolver» объекта DOMConfiguration.

На этапе загрузки и анализа документа LSParser будет вызывать метод resolveResourse() прикрепленного объекта LSResourseResolver для получения объекта типа LSInput, который позволит объекту LSParser получить внешнюю сущность. В данном случае интерфейс LSInput реализуется специфическим образом.

Поддержка рассмотренных интерфейсов DOMLoad&Save в языке Java обеспечена пакетом org.w3c.dom.ls.

Пример. Программное создание документа и сохранение его в поток.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.w2c.dom.Is.*;
import java.util.*;
import java.text.*;

public class DOMTest {
   public static void main (String [] args) {
      try { // создание пустого документа
         DocumentBuilderFactory documentBuilderFactory =
            DocumentBuilderFactory.newInstance();
         DocumentBuilder documentBuilder =
            documentBuilderFactory.newDocumentBuilder();
         Document document = documentBuilder.newDocument();
/* создаем элементы документа следующей структуры
   <message timestamp = "01.05.2005 21:35:45">
      <head>
         <priority>high</priority>
      </head>
      <body>
         <para>get outta here </para>
      </body>
   </message>   */

         Element rootElement = document.createElement ("message");
         Data dateTime = (new GregorianCalendar()).getTime();
         rootElement.setAttribute (
```

```
            "timestamp", DateFormat.getDateTimeInstance().format(dateTime)
            );
         document.appendChild(rootElement);
         Element headElement = document.createElement("head");
         rootElement.appendChild(headElement);
         Element bodyElement = document.createElement("body");
         rootElement.appendChild(bodyElement);

         Element element = document.createElement("priority");
         headElement.appendChild (element);
         element.appendChild(document.createTextNode("high"));

         element = document.createElement("para");
         BodyElement.appendChild(element);
         element.appendChild(document.createTextNode("get outta here"));

// Вывод сгенерированного документа на экран (байтовый поток вывода)

         DOMImplementationLS domImplementationLS =
            (DOMImplementationLS)documentBuilder.getDOMImplementation();
         LSSerializer lsSerializer =
            domImplementationLS.createLSSerializer();
         LSOutput lsOutput = domImplementationLS.createLSOutput();
         lsOutput.setByteStream(System.out);   //*
         lsSerializer.write(rootElement, lsOutput);
      } catch (Exception e) { }
   }
}
```

* - для демонстрации вывода с символьный поток можно заменить помеченную строку на следующую

```
StringWriter stringWriter = new StringWriter();
lsOutput.setCharacterStream(stringWriter);
```

и в конец добавить:

```
System.out.println(stringWriter);
```

Лекция 7. SAX-анализ

SAX-АНАЛИЗ

Базовые классы и интерфейсы для работы с SAX-анализом расположены в пакетах:

1) `org.xml.sax` – содержит основные API (Core SAX API)
2) `org.xml.sax.helpers` – дополнительные классы, которые содержат реализацию по умолчанию для некоторых основных интерфейсов SAX
3) `org.sax.ext` – стандартные SAX-расширения, которые позволяют задействовать некие дополнительные возможности, необязательные для реализации.

Кроме того, спецификация JAXP определяет дополнительные классы в пакете `javax.xml.parsers`, которые позволяют более удобным образом выполнять конфигурирование SAX-анализатора.

Любой SAX-анализатор должен реализовать интерфейс `org.xml.sax.XMLReader`, который позволяет:

1. сконфигурировать SAX-анализатор (методами `getProperty()`/`setProperty()`, `getFeature()`/`setFeature()`)

2. связать с SAX-анализатором объекты-обработчики событий. Все события, генерируемые SAX-анализатором в ходе анализа документа, разделяются на несколько групп. Для каждой из них в Core SAX API предусмотрен свой интерфейс, который должен быть реализован объектом-обработчиком событий соответствующей группы (методы `getContentHandler()`/`setContentHandler()`, `getDTDHandler()`/`setDTDHandler()`, `getEntityResolver()`/`setEntityResolver()`, `getErrorHandler()`/`setErrorHandler()`)

3. выполнить анализ документа (метод `parse()`)

Интерфейс `XMLReader` реализуется некоторым классом, имя которого прикладной программе заранее может быть неизвестно: здесь предусмотрен механизм подобный тому, что используется при работе с DOM-анализатором, который позволяет без перекомпиляции программы использовать различные реализации SAX-анализатора. Для получения объекта этого класса прикладные программы используют фабрику `org.xml.sax.helpers.XMLReaderFactory`, представляющую два стандартных фабричных метода: `createXMLReader()` и `createXMLReader(`*`имя_класса`*`)`. Один из них, без параметров, позволяет выполнить поиск необходимого класса по определенному алгоритму, а второй – создает объект класса, имя которого передано в качестве параметра.

Поиск реализации SAX-анализатора выполняется по следующему алгоритму:

1. проверяется наличие системного свойства с именем `org.xml.sax.driver`. Если оно установлено, то его значение используется как имя класса, реализующего интерфейс `XMLReader`.

3. просматриваются все доступные виртуальной машине во время выполнения приложения JAR-архивы в поисках файла `META-INF/Services/org.xml.sax.driver`, содержащего имя класса, реализующего интерфейс `XMLReader`

3. создается `XMLReader` по умолчанию, входящий в комплект поставки среды выполнения

4. иначе, делается попытка создать экземпляр класса, реализующего устаревший интерфейс `org.xml.sax.Parser` по умолчанию, а также создается экземпляр класса `org.xml.sax.helpers.ParserAdapter`, который позволяет работать с объектом типа `org.xml.sax.XMLReader` – `ParserAdapter` транслирует вызовы стандарта SAX2 в вызовы стандарта SAX1.

Если поиск закончился неудачно, метод `createXMLReader()` выбрасывает `SAXException`.

При конфигурировании экземпляра `XMLReader` методом `setFeature()` в качестве параметра указываются идентификаторы с префиксом `http://xml.org/sax/features/` и некоторым суффиксом, называемым `featureID`. Для каждого из них определен тип доступа (чтение/запись), значение по умолчанию и назначение.

Описание `featureID` включено в описание пакета `org.xml.sax`. В табл. 1 приведены некоторые `featureID`:

Идентификатор (`featureID`)	Доступ	Значение по умолчанию	Описание
`external-general-entities`	r/w	Не определено	Обрабатывать общие внешние сущности
`external-parameter-entities`	r/w	Не определено	Обрабатывать внешние параметризованные сущности
`namespaces`	r/w	`True`	Доступ к идентификатору пространств имен и локальным частям имен
`namespace-prefixes`	r/w	`false`	Доступ и именам с префиксом
`resolve-dtd-uris`	r/w	`True`	Преобразовывать системные идентификаторы в ссылках на DTD в абсолютные
`validation`	r/w	Не определено	Выполнять валидацию или нет; если да – все внешние сущности будут загружены
`xml-1.1`	r	-	true – если поддерживается xml 1.1; false – если xml 1.0

Замечание: при установке значений для `feature` используются значения типа `boolean`. Метод `setFeature()` может выбросить `SAXNotSupportedException` – если указанное значение не может быть установлено; `SAXNotRecognizedException` – если feature с указанным именем не поддерживается

При конфигурировании экземпляра `XMLReader` методом `setProperty()` в качестве параметра указывается идентификатор с префиксом `http://xml.org/sax/properties/` и суффиксом `propertyID`. Для каждого из них определяется тип доступа и назначение. Описание `propertyID` включено в описание пакета `org.xml.sax`. В табл. 2 приведены некоторые `propertyID`:

Идентификатор	Доступ	Описание
`declaration-handler`	r/w	Содержит объект, реализующий интерфейс `org.xml.sax.ext.DeclarationHandler` для тонкой обработки DTD-описания, которая не может быть обеспечена средствами стандартного обработчика
`dom-node`	r/w	Объект типа `org.w3c.dom.Node` – корень некоторого DOM-

		дерева, которое будет анализироваться sax-парсером. В этом случае параметры метода `parse()` будут проигнорированы
`lexical-handler`	r/w	Объект, реализующий интерфейс `LexicalHandler` для обработки некоторых событий, связанных с комментариями, разделителями секции CDATA, началом и окончанием DTD-описания и др, которые не могут быть обработаны стандартными обработчиками

Замечание: При установке значения для свойства используется объект типа `java.lang.Object`. Метод `setProperty()` выбрасывает те же исключения, что и метод `setFeature()`.

Метод `parse()` имеет две версии:

1. с параметром типа `String`, который указывает системный идентификатор, с которого будет считан анализируемый документ

2. с параметром типа `org.xml.sax.InputSource`. В виде объектов этого типа принято представлять источники анализируемых документов.

Объект типа `InputSource` является тонкой оболочкой над символьным, байтовым потоками или системным идентификатором. При этом класс `InputSource` с символьным потоком работает через объект типа `java.io.Reader`, с байтовым потоком – `java.io.InputStream`, а системный идентификатор представляется в виде строки (`String`). Соответственно предусмотрено несколько конструкторов с параметром соответствующего типа и конструктор без параметра, а также методы доступа к объектам ввода, оболочкой над которыми является объект `InputSource`. Для символьного потока: `getCharacterStream()/setCharacterStream()`. Для байтового потока: `getByteStream()/setByteStream()`. Для системного идентификатора: `getSystemId()/setSystemId()`.

При использовании байтового потока можно указать кодировку символом. Для работы с кодировкой существуют методы: `getEncoding()/setEncoding()`. При использовании символьного потока она игнорируется.

В любом случае используется только один из источников (в порядке приоритета):
1. символьный поток
2. байтовый поток
3. системный идентификатор

ИНТЕРФЕЙСЫ ОБРАБОТКИ СОБЫТИЙ

В Core SAX API предусмотрено 4 основных интерфейсов обработки событий, генерируемых SAX-анализатором:

1. `org.xml.sax.ContentHandler` – для обработки содержимого документа

2. `org.xml.sax.DTDHandler` – для обработки событий, относящихся к DTD-документа

3. `org.xml.sax.EntityResolver` – для представления возможности программе самостоятельно преобразовать ссылку на внешнюю сущность (включая внешние DTD и внешние параметризованные сущности), внедренную в документ

4. `org.xml.sax.ErrorHandler` – для обработки ошибок, возникающих в ходе SAX-анализа.

ИНТЕРФЕЙС CONTENTHANDLER

Предусмотрены методы для обработки следующих событий:

1. начало и окончание анализа документа (`startDocument()`/`endDocument()`)

2. начало и окончание элемента (`startElement()`/`endElement()`). Метод `startElement()` принимает следующие параметры: идентификатор пространства имен, локальная часть имени, квалифицированное имя с префиксом, атрибут в виде объекта типа `org.xml.sax.Attributes`.

На значение параметров влияет установка следующих features: `http://xml.org/sax/features/namespaces/namespaces-prefixes`

1. идентификатор пространства имен и локальная часть имени – непустая строка, если `namespaces==true`, иначе может быть пустой строкой

2. квалифицированное имя – непустая строка, если `namespaces-prefix==true`, иначе – может быть пустой строкой (по умолчанию)

Метод `endElement()` принимает только первые три из этих параметров.

3. инструкция обработки (метод `processingInstruction()`). В качестве параметров передаются компоненты target и data инструкции обработки

4. наличие пробельного материала, который может быть проигнорирован, и наличие символьных данных (методы: `ignorableWhitespace()`, `characters()`). Оба метода принимают три параметра:

1. массив символов – загруженный фрагмент анализируемого документа

2, 3. смещение и длина в этом массиве, которые указывают фрагмент документа, с которым связано событие.

Эти методы не должны обращаться к символу массива, за пределами указанного диапазона.

Метод `ignorableWhitespace()` используется парсером обязательно, если он работает в режиме валидации, иначе – необязательно, и вместо него может вызываться метод `characters()`.

5. начало и окончание области действия префикса, назначенного идентификатором пространства имен (методы `startPrefixMapping()`/`endPrefixMapping()`). Методу

`startPrefixMapping()` передается два параметра: префикс и идентификатор пространства имен. Методу `endPrefixMapping()` передается только первый параметр.

6. пропуск ссылки на внешнюю сущность (`skippingEntity()`). Если парсер работает в невалидирующем режиме, то он может пропускать ссылки на внешние сущности и DTD. Вне зависимости от режима на этот процесс может влиять значение следующих features: `http://xml.org/sax/features/external-general-entities`, `http://xml.org/sax/features/external-parameter-entities`

В качестве параметра передается имя сущности. Для параметризованных сущностей имя включает `%`; для ссылки на внешние DTD указывается `[dtd]`.

ИНТЕРФЕЙС DTDHANDLER

Предусматривает методы для обработки событий, связанных с нотациями и объявлениями внешних неанализируемых сущностей. Методы:

1. `notationDecl()` в качестве параметра передаются имя, публичный и системный идентификатор нотации.

2. `unparsedEntityDecl()` в качестве параметра передаются имя, публичный и системный идентификатор сущности, а также имя связанной с ней нотации.

ИНТЕРФЕЙС ENTITYRESOLVER

Содержит метод `resolveEntity()`, который вызывается для обработки ссылок на внешние сущности. В качестве параметров методу передаются публичный и системный идентификаторы внешней сущности, ссылка на которую обрабатывается. Метод должен вернуть анализатору объект типа `InputSource`, который тот будет использовать для считывания внешней сущности. Метод может вернуть специфический `InputSource`, позволяющий загружать внешние сущности, например, из БД.

ИНТЕРФЕЙС ERRORHANDLER

Представляет методы, позволяющие приложению получать уведомление от анализатора о возникающих в ходе анализа документа ошибках.

Если обработчик ошибок не назначен приложением, то все нефатальные ошибки никак не будут обрабатываться, а для фатальных анализатор будет выбрасывать `SAXParserException`.

Если обработчик ошибок назначен, то анализатор, вместо выброса исключения будет вызывать соответствующие методы этого объекта, в том числе и для фатальных ошибок.

Методы обработки будут вызываться только для ошибок, связанных с обработкой XML-документа. Для других ошибок (например, ввода/вывода) анализатор будет выбрасывать соответствующее исключение (например, `IOException`), а методы обработки вызываться не будут. Кроме того, после обнаружения и обработки фатальной ошибки анализ может быть прекращен анализатором.

Предусмотрены следующие методы:

1. `warning()` – уведомляет приложение о предупреждениях, которые не являются ошибками
2. `error()` – уведомляет о нефатальной ошибке, которую приложение может попытаться преодолеть
3. `fatalError()` – уведомляет о непреодолимой ошибке, при обнаружении которой, в большинстве случаев, анализ не может быть продолжен

В качестве параметра все методы принимают информацию в виде объекта типа `SAXParseException` и возвращает `void`.

Любой из этих методов может выбросить `SAXException` для прекращения дальнейшего анализа.

КЛАСС DEFAULTHANDLER

Предусмотрен стандартный класс, который обеспечивает реализацию по умолчанию для рассмотренных 4-х интерфейсов: `org.xml.sax.helpers.DefaultHandler`. Методы всех реализованных им интерфейсов ничего не делают. Метод `resolveEntity()` возвращает `null` (в этом случае анализатор для загрузки внешней сущности использует ее системный идентификатор). Метод `fatalError()` выбрасывает `SAXParseException`.

Приложение использует этот класс как базовый для собственных классов обработчиков событий, переопределяя при этом только те методы базового класса, которые будут обрабатывать необходимые приложению события. Это избавляет от необходимости реализовывать все методы, большинство из которых может быть и не нужно конкретному приложению.

ИНТЕРФЕЙС ATTRIBUTES

Объекты типа `Attributes` используются анализатором для передачи информации об атрибутах элементов обработчику типа `ContentHandler` как параметр вызова метода `startElement()`. Данный интерфейс позволяет получать информацию об атрибуте по индексу. Методы:

1. `getURI()` – возвращает идентификатор пространства имен

2. `getLocalName()` – получить локальную часть имени

3. `getQName()` – получить квалифицированное имя

4. `getType()` – возвращает тип атрибута в виде строки в соответствии со спецификацией XML 1.0 (например, “`CDATA`”, “`ID`”, “`ENTITY`”, …) если для атрибута доступно его DTD-описание. Для атрибутов перечислимого типа возвращается значение “`NMTOKEN`”, если тип атрибута не известен – возвращается “`CDATA`”.

5. `getValue()` – получить значение атрибута.

У последних 2-х методов существуют перегруженные версии, которые позволяют получить информацию не по индексу, а по квалифицированному имени или паре «идентификатор пространства имен - локальная часть имени».

Для получения индекса по `QName` или паре «идентификатор – локальная часть» существует метод `getIndex()`.

Метод `getLength()` возвращает количество атрибутов. Список атрибутов не включает те, что были объявлены в DTD с опцией `#IMPLIED` и не были указаны в открывающем тэге. Также в список не включаются атрибуты, используемые для объявления префикса, если feature `namespace-prefixes` не установлено в `true`.

ИНТЕРФЕЙС LOCATOR

Объект типа `Locator` создается и связывается с объектом обработки событий типа `ContentHandler` методом `setDocumentLocator()` – это выполняет анализатор. Объект обработки событий, реализованный разработчиком приложения, может сохранить переданный объект и использовать его затем в методах обработки событий для получения информации о том, в каком месте документа возникло соответствующее событие. Для этого используются методы:

1. `getLineNumber()`, `getColumnNumber()` – возвращают номер строки и колонки, где возникло событие

2. `getSystemId()`, `getPublicId()` – позволяют получить системный и публичный идентификатор анализируемого документа.

ПОДДЕРЖКА SAX В JAXP

В JAXP предусмотрена пара классов, которые облегчают создание и конфигурирование SAX-анализаторов. Они находятся в пакете `javax.xml.parsers`.

В качестве фабричного класса используется `SAXParserFactory`, назначение и использование которого аналогично тем, что и для класса `DocumentBuilderFactory`. Для создания экземпляра класса – статический метод `newInstance()`.

Для создания экземпляра SAX-анализатора предназначен метод фабрики `newSAXParser()`.

Сам SAX-анализатор представлен классом `SAXParser`, который является тонкой оболочкой над классом, реализующим интерфейс `XMLReader`, - на этапе выполнения используется предварительно сконфигурированный экземпляр `XMLReader` в соответствии с установками, сделанными для `SAXParserFactory`, что избавляет от необходимости индивидуального конфигурирования каждого экземпляра путем установки features и properties по их идентификаторам.

При работе с JAXP используют именно `SAXParserFactory` и `SAXParser`, а не непосредственно `XMLReaderFactory` и `XMLReader`. Это позволяет работать в таком же стиле с SAX-анализатором, что и при работе с DOM-анализатором.

В классе `SAXParserFactory` предусмотрены следующие методы конфигурирования: `setNamespaceAware()`, `setValidating()`, `isNamespaceAware()`, `isValidating()`, которые аналогичны методам класса `DocumentBuilderFactory`.

Для непосредственной работы с features предусмотрены методы: `setFeature()`, `getFeature()`, которые аналогичны методам интерфейса `XMLReader`.

Класса `SAXParser`, использующий внутри для реализации анализа экземпляр `XMLReader`, содержит множество перегруженных версий метода `parse()`, которые позволяют считать анализируемый документ из следующих источников:
1. файл (`java.io.File`)
2. бинарный поток ввода (`java.io.InputStream`)
3. URL, указанный строкой (`String`)
4. источник типа `InputSource`

В качестве объекта обработчика событий указывается экземпляр `DefaultHandler` или его подкласса. Для работы с properties предусмотрены методы: `getProperties()`, `setProperties()`, аналогичные тем, что имеются в `XMLReader`.

Существует метод `getXMLReader()`, который позволяет получить экземпляр `XMLReader`, оболочкой для которого выступает данный экземпляр `SAXParser`.

SAX PLUGABILITY

Механизм полностью аналогичен механизму DOM Plugability, только вместо поиска неабстрактного подкласса `DocumentBuilderFactory` выполняется поиск неабстрактного подкласса класса `SAXParserFactory`. Поиск также включает в себя 4 этапа:

1. анализ системного свойства `javax.xml.parsers.SAXParserFactory`

2. поиск свойства с этим именем в файле `jaxp.properties` в подкаталоге `lib` каталога установки JRE

3. просматриваются все доступные виртуальной машине во время выполнения приложения JAR-архивы в поисках файла `META-INF/Services/javax.xml.parsers.SAXParserFactory`, содержащего имя подкласса, объекты которого фактически создаются

4. используется класс, специфический для данной реализации виртуальной машины.

При работе механизма также возможны исключения `FactoryConfigurationError`, `ParserConfigurationException`, возникающие в аналогичных ситуациях.

ПРИМЕР ПРАКТИЧЕСКОГО ИСПОЛЬЗОВАНИЯ SAX

Дан XML-документ, содержащий информацию о маршруте в виде последовательности точек. Для каждой точки указывается ее имя, расстояние до данной точки от предыдущей (0 – для начальной) и время прохождения участка от предыдущей до данной (0 – для начальной).

```
<?xml version=”1.0” encoding=”UTF-8”?>
<route>
   <point>
      <name>A</name>
      <distance>0</distance>
      <time>0</time>
   </point>
   <point>
      <name>B</name>
      <distance>10.5</distance>
      <time>12.3</time>
   </point>
</route>
```

Необходимо, используя SAX-анализ, преобразовать этот XML-документ в HTML-документ, представляющий информацию в виде таблицы с выводом информации о скорости и расстоянии нарастающим итогом.

Point	Distance	Time	Speed	Overall distance
A	0	0	0	0
B	10.5	12.3	51.2	10.5

Класс `ParseRoute`

```
package ru.arriah.labs.sax;

import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ParseRoute {
   public static void main(String[] args) {
      try {
         SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
         SAXParser saxParser = saxParserFactory.newSAXParser();
         DefaultHandler defaultHandler = new RouteHandler();
         saxParser.parse(System.in, defaultHandler);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Класс `RouteHandler`

```
package ru.arriah.labs.sax;

import java.text.*;
import java.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class RouteHandler extends DefaultHandler {
   protected double overallDistance;
   protected boolean isDistance;
   protected boolean isTime;
   protected double distance;
   protected double speed;
   protected NumberFormat numberFormat;

   public RouteHandler() {
```

```
      super();
      overallDistance = 0;
      isDistance = false;
      isTime = false;
      numberFormat = NumberFormat.getInstance(new Locale("ru", "RU"));
      numberFormat.setMaximumFractionDigits(3);
   }

   public void startDocument() throws SAXException {
      System.out.println("<html>");
      System.out.println("   <head>");
      System.out.println("              <link  href  =  \"route.css\"  rel  =
\"stylesheet\" type = \"text/css\">");
      System.out.println("   </head>");
      System.out.println("   <body>");
      System.out.println("      <table>");
      System.out.println("         <thead>");
      System.out.println("            <tr>");
      System.out.println("               <th>Point</th>");
      System.out.println("               <th>Distance</th>");
      System.out.println("               <th>Time</th>");
      System.out.println("               <th>Speed</th>");
      System.out.println("               <th>Overall distance</th>");
      System.out.println("            </tr>");
      System.out.println("         </thead>");

      System.out.println("         <tbody>");
   }

   public void startElement(
      String uri, String localName, String qName, Attributes attributes
   ) throws SAXException {
      if (qName.equals("point")) {
         System.out.println("            <tr>");
      } else if (!qName.equals("route")) {
         System.out.print("               <td>");
         isDistance = qName.equals("distance");
         isTime = qName.equals("time");
      }
   }

   public void endElement(
      String uri, String localName, String qName
   ) throws SAXException {
      if (qName.equals("point")) {
         System.out.println("                                          <td>"   +
numberFormat.format(speed) + "</td>");
         System.out.println("                                          <td>"   +
numberFormat.format(overallDistance) + "</td>");
         System.out.println("");
         System.out.println("            </tr>");
      } else if (!qName.equals("route")) {
         System.out.println("</td>");
      }
   }

   public  void  characters(char[]  ch,  int  start,  int  length)  throws
SAXException {
      String s = new String(ch, start, length);
      s = s.trim();

      if (s.length() > 0) {

         if (isDistance) {
```

```
            try {
               distance = Double.parseDouble(s);
               s = numberFormat.format(distance);
               overallDistance += distance;
            } catch (Exception e) {
               throw new SAXException(e);
            }
         } else if (isTime) {
            try {
               double time = Double.parseDouble(s);
               s = numberFormat.format(time);
               speed = distance/time;
            } catch (Exception e) {
               throw new SAXException(e);
            }
         }

         System.out.print(s);
      }
   }

   public void endDocument() throws SAXException {
      System.out.println("         </tbody>");
      System.out.println("      </table>");
      System.out.println("   </body>");
      System.out.println("</html>");
   }
```

Лекция 8. StAX-анализ

StAX-АНАЛИЗ

StAX состоит из двух наборов API для обработки XML, которые обеспечивают разные уровни абстракции. API с использованием курсора позволяет приложениям работать с XML как с потоком лексем (или событий); приложение может проверить статус анализатора и получить информацию о последней проанализированной лексеме, а затем перейти к следующей. Это, скорее, низкоуровневый API; он довольно эффективный, но не предоставляет абстракции нижележащей XML-структуры. Высокоуровневый API, использующий итераторы событий, позволяет приложению обрабатывать XML как серию объектов событий, каждый из которых соответствует фрагменту XML-структуры. Все, что требуется от приложения - это определить тип синтаксически разобранного события, отнести его к соответствующему конкретному типу и использовать соответствующие методы для получения информации, относящейся к событию.

Базовые классы для StAX-анализатора расположены в пакете `javax.xml.stream`. Стандартные события определены в пакете `javax.xml.stream.events`.

Для создания StAX-анализатора предназначена фабрика `XMLInputFactory`. Создание экземпляра фабрики осуществляется статическими методами `newInstance()`.

Для конфигурирования фабрики используются свойства, наиболее важные из которых представлены в табл. 1. Названия всех свойств начинаются на "`javax.xml.stream.`", поэтому данная часть названия опущена. Все перечисленные в таблице свойства имеют тип `Boolean`.

Название свойства	Поведение	По умолчанию
`isValidating`	Валидация по DTD.	False
`isNamespaceAware`	Обработка пространств имен	True
`isCoalescing`	Слияние смежных текстовых узлов	False
`isReplacingEntityReferences`	Заменять ссылки на внутренние сущности замещающим текстом	True
`isSupportingExternalEntities`	Обработка внешних анализируемых сущностей	Не указано
`supportDTD`	Представление DTD в виде события	True

Чтобы работать с API с использованием курсора, приложение получает объект `XMLStreamReader` посредством вызова методов `createXMLStreamReader()`. Чтобы работать с API с использованием итератора событий, приложение вызывает один из методов `createXMLEventReader()`, чтобы создать объект типа `XMLEventReader`.

Оба интерфейса – и `XMLStreamReader`, и `XMLEventReader` – позволяют приложению самому выполнять итерацию по нижележащему XML-потоку. Различие между двумя подходами заключается в том, как каждый из них представляет фрагменты разобранного XML-документа.

Интерфейс `XMLStreamReader` действует как курсор, который размещается сразу после последней разобранной XML-лексемы и предоставляет методы для получения

информации о ней. Этот подход очень эффективно использует память, так как не создает новых объектов.

Интерфейс `XMLEventReader`, по сути, представляет собой стандартный Java-итератор, который преобразует XML в поток объектов событий. Каждый объект события, в свою очередь, инкапсулирует информацию, имеющую отношение к конкретной XML-структуре, которую он представляет.

Какой из стилей API использовать, зависит от ситуации. API с использованием итераторов событий является более объектно-ориентированным подходом, чем API с использованием курсора. По существу, его проще использовать в модульной архитектуре, поскольку текущее состояние синтаксического анализатора отражается в объекте события; следовательно, компоненту приложения не нужен доступ к анализатору/считывателю при обработке события. Более того, `XMLEventReader` можно создать из `XMLStreamReader` при помощи метода объекта `XMLInputFactory createXMLEventReader(XMLStreamReader)`.

Подход с использованием итератора событий предлагает существенное преимущество по сравнению с методом, в котором используется курсор. Посредством превращения событий синтаксического анализатора в объекты первого уровня, он позволяет приложению обрабатывать их объектно-ориентированным способом. Это способствует улучшенной модульности и многократному использованию кода в нескольких компонентах приложения. По этой причине далее будет рассмотрен только API с использованием итератора событий.

ИНТЕРФЕЙС XMLEVENTREADER

Основной интерфейс API с использованием итератора событий – `XMLEventReader`, расширяющий `java.util.Iterator`. Вся информация об анализируемом событии инкапсулирована не в считывателе, а в объекте события.

Класс `XMLInputFactory` поддерживает следующие источники ввода, которые можно использовать для создания `XMLEventReader`:
1) бинарные и символьные потоки ввода;
2) `javax.xml.transform.Source` (из TrAX), который способствует интеграции StAX в API преобразования JAXP (TrAX);
3) `XMLStreamReader`.

После создания считывателя событий `XMLEventReader` приложение может использовать его для выполнения итераций на событиях, которые представляют собой фрагменты XML-документа. Поскольку интерфейс `XMLEventReader` является расширением `java.util.Iterator`, то можно использовать стандартные методы, такие как `hasNext()` и `next()`. При этом метод `remove()` не поддерживается, при его вызове выбрасывается исключение.

`XMLEventReader` предоставляет также некоторые удобные методы для упрощения обработки XML:

1. `nextEvent()` – типизированный вариант метода `next()`, он возвращает `XMLEvent`, базовый интерфейс всех объектов событий;

2. `nextTag()` – пропускает все незначимые пробелы вплоть до следующего открывающего или закрывающего тэга. Возвращаемым значением будет либо событие `StartElement`, либо `EndElement`;
3. `getElementText()` – возвращает текстовое содержимое чисто текстового элемента. Начиная с `StartElement` как текущего события, метод выполняет конкатенацию всех символов и возвращает строку до `EndElement`;
4. `peek()` – распознает следующее событие (если таковое имеется), без перемещения указателя потока.

Приложение может использовать метод `getProperty(String)` для получения значения свойства StAX-анализатора.

По завершении анализа приложение должно завершить работу анализатора, вызвав его метод `close()`, чтобы высвободить все ресурсы, которые были задействованы в этом процессе.

СОБЫТИЯ StAX-АНАЛИЗА

Стандартные типы объектов событий определены в пакете `javax.xml.stream.events`. Интерфейс `XMLEvent` представляет корень иерархии; все типы событий должны быть расширениями этого интерфейса. Можно определять пользовательские события с учетом данного требования.

После получения события от анализатора приложению обычно приходится отнести его к одному из подтипов `XMLEvent`, чтобы получить доступ к специфической для типа информации этого события. Существует несколько способов выполнить эту задачу. Кроме проверки экземпляра события операцией `instanceof`, интерфейс `XMLEvent` предоставляет метод `getEventType()`, который возвращает тип события в виде константы из класса `XMLStreamConstants`. Например, если метод `getEventType()` события возвращает `START_ELEMENT`, то это событие может быть с уверенностью отнесено к типу `StartElement`.

Еще один способ определить конкретный тип события – это использовать один из методов-запросов, имеющихся для этой цели. Например, метод `isStartElement()` возвращает значение `true`, если это `StartElement` и так далее.

Для приведения события к типам `StartElement`, `EndElement` и `Characters` предназначены методы `asStartElement()`, `asEndElement()` и `asCharacters()` , соответственно.

Также в интерфейсе `XMLEvent` определены методы `getLocation()`, `getSchemaType()` и `writeAsEncodedUnicode(Writer)`. Метод `getLocation()` возвращает объект `Location` с информацией о размещении события в источнике ввода. Метод `getSchemaType()` извлекает информацию о типе в XML-схеме, относящуюся к данному событию. Метод `writeAsEncodedUnicode(Writer)` записывает событие в символьный поток вывода в соответствии со спецификацией XML.

XML-ДОКУМЕНТ

При выполнении синтаксического анализа потока, который представляет XML-документ, первым событием, которое возвращает `XMLEventReader`, будет `StartDocument`. Этот

интерфейс предоставляет методы для получения информации о самом документе. Например, метод `getSystemId()` возвращает системный идентификатор документа, если он известен. Метод `getVersion()` возвращает версию XML, используемую в этом документе.

Метод `getCharacterEncodingScheme()` возвращает информацию о кодировке документа, либо явно заданную в XML-декларации, либо распознанную автоматически синтаксическим анализатором. По умолчанию установлено значение UTF-8. Метод `isStandalone()` возвращает значение `true`, если не существует внешнего описания разметки или значение не определено явно в XML-декларации документа.

Последнее событие, выдаваемое считывателем `XMLEventReader` — это `EndDocument`, представляющее конец XML-документа. Это событие не определяет новых методов.

DTD, ОБЪЯВЛЕНИЯ СУЩНОСТЕЙ И НОТАЦИЙ

Если `XMLEventReader` обнаруживает DTD, то он возвращает его как событие `DTD`. Если приложению не нужна информация о DTD, то оно может выполнить запрос на отключение такого поведения синтаксического анализатора посредством установки значения свойства `javax.xml.stream.supportDTD` в `false`. Метод события `getDocumentTypeDeclaration()` может извлечь всю информацию, имеющуюся в DTD, одной строкой, включая внутренние подмножества. Метод `getEntities()` возвращает список событий `EntityDeclaration`, которые представляют общие внешние и внутренние сущности. Метод `getNotations()` возвращает список событий `NotationDeclaration`, представляющих нотации, объявленные в DTD.

Событие `EntityDeclaration` представляет собой общую сущность, объявленную в DTD. Это событие можно получить только в составе события `DTD`. Оно предоставляет методы для получения имени объекта, его открытого и системного идентификатора, а также имени нотации (методы `getName()`, `getPublicId()`, `getSystemId()` и `getNotationName()`, соответственно). Для внутренней сущности метод `getReplacementText()` возвращает ее содержимое.

Событие `NotationDeclaration` представляет объявление нотации и также доступно только через событие `DTD`. Кроме имени (метод `getName()`), этот интерфейс предоставляет методы для получения публичного и системного идентификатора (методы `getPublicId()` и `getSystemId()`, соответственно).

ЭЛЕМЕНТЫ, АТРИБУТЫ И ОБЪЯВЛЕНИЯ ПРОСТРАНСТВ ИМЕН

Для каждого элемента `XMLEventReader` возвращает событие `StartElement` для представления его открывающего тэга, и соответствующее событие `EndElement` для представления закрывающего тэга. Даже для пустых элементов, у которых нет открывающего и закрывающего тэгов (например, элемент `<empty/>`), считыватель возвращает событие `StartElement`, и сразу вслед за этим - событие `EndElement`.

Событие `StartElement` обычно используется для представления большей части информации в XML-документе. Метод `getName()` возвращает квалифицированное имя элемента в виде объекта класса `QName`, который инкапсулирует все компоненты квалифицированного имени, такие как URI пространства имен, префикс и локальное имя.

Метод `getNamespaceContext()` позволяет извлечь информацию о текущем контексте пространства имен вместе с информацией обо всех пространствах имен, которые в настоящее время используются в области действия. Для извлечения атрибутов элемента предназначен метод `getAttributes()`, для получения отдельных атрибутов по их имени – метод `getAttributeByName(QName)`. Получить все пространства имен, объявленные в элементе, можно с помощью метода `getNamespaces()`. Метод `getNamespaceURI(String)` возвращает пространство имен, обозначенное указанным префиксом в актуальном контексте.

Хотя атрибуты элементов моделируются как события и представлены интерфейсом `Attribute`, доступ к ним можно получить только через событие `StartElement`. Метод `getName()` возвращает квалифицированное имя элемента, а метод `getValue()` – его значение в виде строки. Метод `isSpecified()` вызывается для того, чтобы определить, действительно ли данный атрибут задан для элемента, или он применяется схемой документа. Метод `getDTDType()` возвращает объявленный тип атрибута (то есть, CDATA, IDREF или NMTOKEN).

Объявленные в элементе пространства имен доступны в виде событий `Namespace` только из события `StartElement`. Интерфейс `Namespace` является расширением интерфейса `Attribute`, поскольку пространства имен, по сути, определяются как атрибуты некоторого элемента с особым префиксом. Метод `getPrefix()` позволяет получить префикс пространства имен (для пространства имен по умолчанию возвращается пустая строка). Метод `getNamespaceURI()` возвращает URI пространства имен. Метод `isDefaultNamespaceDeclaration()` определяет, является ли данное пространство имен пространством имен по умолчанию.

Событие `EndElement` представляет закрывающий тэг элемента (или просто окончание разметки элемента, если это пустой элемент). Метод этого события `getName()` можно использовать для получения квалифицированного имени, а метод `getNamespaces()` – для того, чтобы узнать, какие пространства имен вышли из области действия.

ТЕКСТОВОЕ СОДЕРЖИМОЕ

Событие `Characters` используется для представления трех типов текстовых событий: текст, имеющий реальное содержимое (CHARACTERS), разделы CDATA и игнорируемые пробелы (SPACE). Событие предоставляет методы для различения этих трех подтипов; метод `isCData()` возвращает значение `true`, если данные являются событием CDATA, а метод `isIgnorableWhitespace()` – если это событие SPACE. Метод `getData()` возвращает текст события. Метод `isWhiteSpace()` показывает, состоит ли текст из одних пробельных символов.

Событие `EntityReference` возвращается для ссылок на сущности. Для анализируемых сущностей оно возвращается только в том случае, если свойство анализатора `javax.xml.stream.isReplacingEntityReferences` установлено в `false`. В противном случае синтаксическому анализатору требуется заменить ссылки на внутренние сущности замещающим их текстом и представить их как обычные символьные события или проанализировать внешние сущности и представить их как обычную разметку. Интерфейс `EntityReference` предоставляет методы `getName()` и `getDeclaration()` для получения имени сущности и ее объявления (события `EntityDeclaration`).

Инструкции обработки и комментарии представлены, соответственно, интерфейсами `ProcessingInstruction` и `Comment`. Интерфейс `ProcessingInstruction` предоставляет методы `getTarget()` и `getData()` для извлечения назначения инструкции и данных. Метод `getText()`, определенный в интерфейсе `Comment`, возвращает текст комментария.

ФИЛЬТРАЦИЯ СОБЫТИЙ И МАНИПУЛИРОВАНИЕ ПОТОКОМ СОБЫТИЙ

Можно создать фильтрующий анализатор `XMLEventStream` для ситуаций, в которых приложение ожидает поток определенных событий. Для этого предназначен метод фабрики анализаторов `createXMLEventReader(XMLEventReader, EventFilter)`. Фильтр (объект типа `EventFilter`) в методе `accept(XMLEvent)` принимает или отклоняет события, полученные от анализатора.

Для выполнения более сложных манипуляций потоком необходимо расширить вспомогательный класс `EventReaderDelegate`, определенный в пакете `javax.xml.stream.util` и реализующий интерфейс `XMLEventStream`. Этот класс позволяет разработчику упаковать имеющийся анализатор `XMLEventStream`, которому по умолчанию делегируются все вызовы. Этот подкласс может затем отменить любые методы, чтобы изменить поведение базового считывателя. Например, можно использовать этот подход для инъекции пользовательских событий в поток событий или каким-либо иным образом преобразовывать этот поток.

ЗАПИСЬ XML-ДОКУМЕНТОВ

StAX определяет API записи файловых объектов. Этот потоковый API, как и его анализирующий аналог, поставляется в двух разновидностях – низкоуровневый интерфейс `XMLStreamWriter`, который работает с лексемами, и высокоуровневый интерфейс `XMLEventWriter`, который работает с объектами событий. `XMLStreamWriter` предоставляет методы для записи отдельных XML-лексем (таких как тэги открытия и закрытия или атрибуты элементов) без проверки их корректности. Интерфейс `XMLEventWriter`, с другой стороны, позволяет приложению добавлять в вывод события XML.

Независимо от выбранного стиля API, приложение должно сначала использовать фабрику `XMLOutputFactory` для создания соответствующего объекта-писателя. Чтобы получить экземпляр фабрики `XMLOutputFactory` с параметрами по умолчанию, вызывается статический метод `newInstance()`.

Класс `XMLOutputFactory` поддерживает несколько типов вывода, в основном, аналоги типов ввода, поддерживаемых классом `XMLInputFactory`; кроме бинарных и символьных потоков вывода (`OutputStream` и `Writer`), поддерживается также JAXP-тип `Result`.

Интерфейс `XMLEventWriter` использует события для представления фрагментов XML-документа. Метод `add(XMLEvent)` позволяет записать документа от начала до конца. Метод `add(XMLEventReader)` записывает все события, которые он получает от считывателя. При помощи этого метода приложение может эффективно направить содержимое всего XML-потока через канал в другой XML-поток, не изменяя его.

Для удобства программирования события для атрибутов и пространств имен не требуется явно добавлять в событие открывающего тэга `StartElement`. Объект-писатель обязан

запоминать в буфере событие `StartElement`, вплоть до добавления события, отличного от атрибута и пространства имен.

Для управления пространством имен предназначены следующие методы. Методы `getNamespaceContext()` и `setNamespaceContext(NamespaceContext)` обеспечивают доступ ко всему контексту пространства имен. Метод `setPrefix(String, String)` устанавливает, а `getPrefix(String)` возвращает привязки префиксов пространств имен. Метод `setDefaultNamespace(String)` задает пространство имен по умолчанию для текущего контекста пространства имен.

Методы `flush()` и `close()` предназначены для сброса всех кэшированных данных и закрытия объекта-писателя, соответственно. Обратите внимание на то, что метод `close()` только освобождает ресурсы, которые занимал объект-писатель, но не закрывает поток вывода.

Для записи событий приложение должно иметь возможность создавать экземпляры событий. Эти функции предоставляются фабрикой `XMLEventFactory`, которая определяет методы для создания событий всех стандартных типов. Для получения конкретного экземпляра фабрики предназначен статический метод `newInstance()`.

Пример записи простого XHTML-документа:

```
final String XHTML_NS = "http://www.w3.org/1999/xhtml";
final QName HTML_TAG = new QName(XHTML_NS, "html");
final QName HEAD_TAG = new QName(XHTML_NS, "head");
final QName TITLE_TAG = new QName(XHTML_NS, "title");
final QName BODY_TAG = new QName(XHTML_NS, "body");

XMLOutputFactory f = XMLOutputFactory.newInstance();
XMLEventWriter w = f.createXMLEventWriter(System.out);
XMLEventFactory ef = XMLEventFactory.newInstance();
try {
      w.add(ef.createStartDocument());
      w.add(ef.createIgnorableSpace("\n"));
      w.add(ef.createDTD("<!DOCTYPE html " +
                  "PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" " +
                  "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">"));
      w.add(ef.createIgnorableSpace("\n"));
      w.add(ef.createStartElement(HTML_TAG, null, null));
      w.add(ef.createNamespace(XHTML_NS));
      w.add(ef.createAttribute("lang", "en"));
      w.add(ef.createIgnorableSpace("\n"));
      w.add(ef.createStartElement(HEAD_TAG, null, null));
      w.add(ef.createStartElement(TITLE_TAG, null, null));
      w.add(ef.createCharacters("Test"));
      w.add(ef.createEndElement(TITLE_TAG, null));
      w.add(ef.createEndElement(HEAD_TAG, null));
      w.add(ef.createIgnorableSpace("\n"));
      w.add(ef.createStartElement(BODY_TAG, null, null));
      w.add(ef.createCharacters("This is a test."));
      w.add(ef.createEndElement(BODY_TAG, null));
      w.add(ef.createEndElement(HTML_TAG, null));
      w.add(ef.createEndDocument());
} finally {
      w.close();
}
```

ОБРАБОТКА ОШИБОК

Многие методы интерфейсов `XMLEventReader` и `XMLEventWriter` выбрасывают исключения класса `XMLStreamException`, который является базовым классом исключений StAX-анализа. Именно с его помощью анализатор сообщает о нарушениях правил XML в обрабатываемом документе, а также о прочих фатальных ошибках в процессе анализа. В классе исключения `XMLStreamException` определены методы `getLocation()` для получения вызвавшего исключение местоположения в документе и `getNestedException()`, возвращающий вложенное исключение.

Для обработки предупреждений и нефатальных ошибок, возникающих в процессе анализа XML-документа, предназначен интерфейс `XMLReporter`, объект которого связывается с фабрикой потоковых анализаторов методом `XMLInputFactory.setXMLReporter()`. Данная настройка действует для всех анализаторов, порождаемых фабрикой.

В интерфейсе `XMLReporter` определен единственный метод `report()`, которому передаются сообщение об ошибке, тип ошибки (определяется реализацией), связанная с ошибкой информация и местоположение ошибки. Данный метод выбрасывает исключение `XMLStreamException`, что позволяет прерывать анализ XML-документа.

StAX PLUGABILITY

Этот механизм позволяет заменить используемую программой реализацию StAX без перекомпиляции. Механизм основан на том, что метод `newInstance()` классов `XMLInputFactory`, `XMLOutputFactory` и `XMLEventFactory` выполняет поиск неабстрактного подкласса в следующей последовательности:

1. проверяется наличие системного свойства с именем `javax.xml.stream.XMLInputFactory`, `javax.xml.stream.XMLOutputFactory` или `javax.xml.stream.XMLEventFactory`. Если они установлены, то их значения используются как имена классов, объекты которых фактически создаются.

2. проверяется наличие файла `stax.properties` в подкаталоге `lib` каталога установки JRE, содержащего указанные выше свойства

3. просматриваются все доступные виртуальной машине во время выполнения приложения JAR-архивы в поисках файлов `META-INF/Services/javax.xml.stream.XMLInputFactory`, `META-INF/Services/javax.xml.stream.XMLOutputFactory` и `META-INF/Services/javax.xml.stream.XMLEventFactory`, содержащих имя реализации соответствующей фабрики

4. используется подкласс, специфичный для данной реализации виртуальной машины.

Если реализация фабрики не найдена методом `newInstance()`, то выбрасывается исключение `FactoryConfigurationError`.

ПРИМЕР ИСПОЛЬЗОВАНИЯ StAX

Пример вывода в отчете все элементов и атрибутов расширения Atom (то есть таких элементов и атрибутов, которые не принадлежат к пространству имен Atom или к пространству имен XML). Atom - это формат синдикации, используемый для размещения различных материалов в Интернете. Для каждого события `StartElement` проверяется, что URI его пространства имен является URI пространства имен Atom. Затем выполняется итерация по всем атрибутам и используется интерфейс `Attribute` для получения имен. После этого выводятся все атрибуты, которые не принадлежат к пространствам имен Atom или XML. Кроме того, показан пример использования интерфейса `Characters` для вывода различных типов текстового содержимого, а также интерфейсов `Comment` и `ProcessingInstruction`.

```
final String ATOM_NS = "http://www.w3.org/2005/Atom";

URL url = new URL(uri);
InputStream input = url.openStream();
XMLInputFactory f = XMLInputFactory.newInstance();
XMLEventReader r = f.createXMLEventReader(uri, input);
try {
      while (r.hasNext()) {
            XMLEvent event = r.nextEvent();
            if (event.isStartElement()) {
                  StartElement start = event.asStartElement();
                  boolean isExtension = false;
                  boolean elementPrinted = false;
                  if (!ATOM_NS.equals(start.getName().getNamespaceURI())) {
                        System.out.println(start.getName());
                        isExtension = true;
                        elementPrinted = true;
                  }

                  for (Iterator i = start.getAttributes(); i.hasNext();) {
                        Attribute attr = (Attribute) i.next();
                        String ns = attr.getName().getNamespaceURI();
                        if (ATOM_NS.equals(ns))
                              continue;

                        if (".equals(ns) & !isExtension)
                              continue;

                        if ("xml".equals(attr.getName().getPrefix()))
                              continue;

                        if (!elementPrinted) {
                              elementPrinted = true;
                              System.out.println(start.getName());
                        }

                        System.out.print("\t");
                        System.out.println(attr);
                  }
            } else if (event.isCharacters()) {
                  Characters c = event.asCharacters();
                  System.out.print("Characters");
                  if (c.isCData()) {
                        System.out.print(" (CDATA):");
                        System.out.println(c.getData());
                  } else if (c.isIgnorableWhiteSpace()) {
                        System.out.println(" (IGNORABLE SPACE)");
                  } else if (c.isWhiteSpace()) {
```

```
                        System.out.println(" (EMPTY SPACE)");
                    } else {
                        System.out.print(": ");
                        System.out.println(c.getData());
                    }
                } else if (event.isProcessingInstruction()) {
                    ProcessingInstruction pi = (ProcessingInstruction) event;
                    System.out.print("PI(");
                    System.out.print(pi.getTarget());
                    System.out.print(", ");
                    System.out.print(pi.getData());
                    System.out.println(")");
                } else if (event.getEventType() == XMLStreamConstants.COMMENT) {
                    System.out.print("Comment: ");
                    System.out.println(((Comment) event).getText());
                }
            }
        } finally {
            r.close();
        }

        input.close();
```

Лекция 9. Программное XSLT-преобразование

XSLT TRANSFORMATION

Набор API для преобразования XML-документов располагается в пакете `javax.xml.transform` и вложенных в него пакетах. Система классов аналогична `javax.xml.parsers`.

Реализация JAXP 1.4 обязана поддерживать XSLT 1.0 и XPath 1.0, но в настоящее время актуальными версиями являются XSLT 2.0 и XPath 2.0. Существует несколько реализаций JAXP, которые поддерживают последние версии стандартов W3C, например, Saxon 8.9, Oracle XDK. При этом поддержка новой версии языка XSLT не приводит к изменению программных интерфейсов, что позволяет применять в XSLT-документах новые возможности без изменения программного кода, запускающего XSLT-преобразование.

ФАБРИКА TRANSFORMERFACTORY

Преобразование осуществляется объектом типа `Transformer`, который создается фабрикой `TransformerFactory`. Данная фабрика является абстрактной, поэтому экземпляр фабрики создается с помощью статического метода `newInstance()`.

Для создания объекта-трансформатора предназначен метод фабрики `newTransformer()`, который имеет две перегруженные версии:
– без параметров, создает трансформер, выполняющий копирование источника в результат, иными словами – тождественное преобразование (identity transform). Пустой трансформер может быть использован для сохранения DOM-дерева в файл;
– в качестве параметра типа `Source` указывается источник XSLT-документа, по которому выполняется преобразование XML-документа.

Для конфигурирования `TransformerFactory` имеется два механизма.

1. Механизм feature, поддержка которого осуществляется методами `get-/setFeature()`, в основном предназначен для установки свойств, поддерживаемых конкретной реализацией. Feature имеет значение логического типа.
Спецификация требует поддержки feature с именем, задаваемом константой `XMLConstants.FEATURE_SECURE_PROCESSING`. Если значение данной feature – `true`, то преобразование выполняется с учетом ограничений по памяти и безопасности, вследствие чего могут не обрабатываться некоторые конструкции языка XSLT и функции, определенные пользователем.
Если feature не поддерживается, то выбрасывается исключение `TransformerConfigurationException`.

2. Механизм атрибутов, поддержка которого осуществляется методами `get-/setAttribute()`, полностью предназначен для установки свойств, поддерживаемых конкретной реализацией. Атрибут может иметь произвольное значение объектного типа. Если атрибут не поддерживается, то выбрасывается исключение `IllegalArgumentException`.

В спецификациях W3C определен стандартный способ внедрения в XML-документы ссылок на таблицы стилей (к которым относятся и XSLT-документы) при помощи

инструкции обработки `xml-stylesheet`, употребляемой в прологе XML-документа. Пример:

```
<?xml-stylesheet href="transform.xsl" media="print" title="Compact" type="text/xml"?>
```

В фабрике имеется метод `getAssociatedStylesheet()`, который позволяет получить XSLT-документ, на который имеется ссылка в XML-документе, передаваемом в качестве параметра. Метод возвращает XSLT-документ в виде объекта типа `Source`. Для поиска наиболее подходящего XSLT-документа в метод передаются значения атрибутов `media`, `title` и `charset` инструкции обработки `xml-stylesheet`.

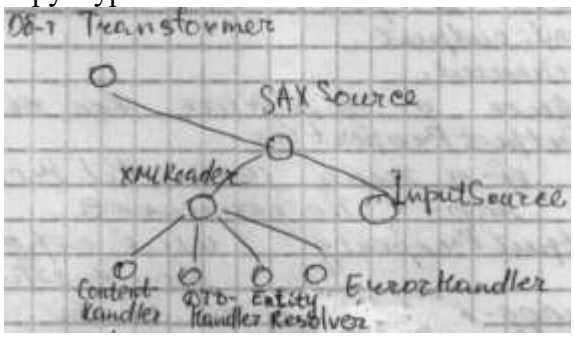
XSLT-ПРЕОБРАЗОВАНИЕ С ПОМОЩЬЮ ОБЪЕКТА TRANSFORMER

Преобразование осуществляется методом `transform()`, который принимает два параметра – преобразуемый XML-документ (объект `Source`) и приемник результата преобразования (объект `Result`). За счет того, что для представления источника и результата существуют базовые интерфейсы, отпала необходимость в перегруженных версиях метода для различных комбинаций источника и результата.

Для интерфейсов `Source` и `Result`, представляющих источник и результат преобразования, предусмотрено четыре стандартных реализации.

1. Пакет `javax.xml.transform.dom` содержит средства для представления DOM-дерева в качестве источника или результата преобразования. В этом пакете определены классы `DOMSource` и `DOMResult`.
Объект `DOMSource` связывается с узлом DOM-дерева (объектом типа `Node`), который считается корневым узлом для выполняемого преобразования. Как правило, в качестве источника используется XML-документ целиком (объект `Document`).
Объект `DOMResult` связывается с узлом DOM-дерева, который является родительским узлом для результирующего поддерева. По умолчанию результат преобразования становится последним вложенным узлом, но можно указать узел DOM-дерева, перед которым нужно вставить результирующее поддерево.
Если использовать конструктор без параметров для `DOMSource` или `DOMResult`, то источником преобразования или корнем результата будет вновь созданный пустой документ.

2. Пакет `javax.xml.transform.sax` содержит средства для представления потока событий SAX в качестве источника (`SAXSource`) или результата преобразования (`SAXResult`).
Для создания объекта типа `SAXSource` достаточно указать источник типа `InputSource`, а также можно привести анализатор типа `XMLReader`. Фактически получается следующая структура объектов:



Приемник результата преобразования типа `SAXResult` связывается с обработчиком событий типа `ContentHandler`. Этот обработчик получает события, связанные с содержимым XML-документа, появляющегося в результате преобразования.

3. Пакет `javax.xml.transform.stax` содержит средства для представления потока событий StAX в качестве источника (`StAXSource`) или результата преобразования (`StAXResult`). В рамках StAX определено два набора API – с использованием курсора и с использованием итератора событий, – и для представления источника и результата преобразования можно использовать любой из них.
Источник `StAXSource` связывается со считывателем лексем или считывателем событий (объект типа `XMLStreamReader` или `XMLEventReader`, соответственно).
Приемник результата `StAXResult` связывается с объектом-писателем лексем или событий (объект типа `XMLStreamWriter` или `XMLEventWriter`, соответственно).

4. Пакет `javax.xml.transform.stream` позволяет использовать потоки ввода/вывода в качестве источника и приемника преобразования (классы `StreamSource` и `StreamResult`, соответственно). Можно использовать в качестве таких потоков объекты следующих типов:
1) текстовые потоки ввода/вывода (`java.io.Reader`, `java.io.Writer`)
2) бинарные потоки ввода/вывода (`java.io.InputStream`, `java.io.OutputStream`)
3) системный идентификатор в виде строки java.lang.String
4) файлы (`java.io.File`)

В качестве источника и приемника преобразования разрешается использовать объекты, использующие различные варианты представления XML-документа. Например, можно использовать SAX-источник и приемник в виде потока вывода для преобразования XML-документа в текстовый файл.

Каждый класс источника и приемника преобразования содержит статическое поле `FEATURE` с именем свойства, которое можно использовать для запроса feature у фабрики трансформеров. Если фабрика поддерживает указанную возможность (метод `getFeature()` возвращает `true`), то она поддерживает источник или приемник соответствующего типа. Пример:

```
if (transformerFactory.getFeature(DOMSource.FEATURE)) {
    // создать DOM-источник
} else if (transformerFactory.getFeature(SAXSource.FEATURE)) {
    // создать SAX-источник
}
```

Язык XSLT требует поддержки пространств имен, поэтому источник преобразования рекомендуется формировать с помощью XML-анализатора, обрабатывающего пространства имен (свойство `isNamespaceAware` является обязательным для всех видов анализаторов, поддерживаемых JAXP). В противном случае выполнение преобразования может завершиться ошибкой. Таким образом, перед созданием источника преобразования следует установить значение `true` для свойства анализатора `isNamespaceAware`.

Конфигурирование трансформера. Параметры преобразования, которые поддерживаются реализацией, можно устанавливать методом `setParameter()`. Имя параметра задается в квалифицированной форме: `{namespace_uri}local_name`.

Из трансформера можно получить параметры вывода (атрибуты элемента `<xsl:output>`) с помощью метода `getOutputProperties()`. Параметры возвращаются в виде объекта `java.util.Properties`, хранящего пары «имя - значение». Полученные параметры можно

изменить и передать трансформеру через метод `setOutputProperties()`. Параметры можно устанавливать и по отдельности методом `setOutputProperty()`. Если какие-либо параметры не поддерживаются, то выбрасывается `IllegalArgumentException`. Имена параметров вывода задаются в квалифицированной форме: `{namespace_uri}local_name`. Для имен всех стандартных параметров вывода определены константы в классе `OutputKeys`. Параметры вывода, установленные программно, имеют приоритет над теми, которые заданы в XSLT-документе.

Для возврата трансформера в исходное состояние используется метод `reset()`.

ПРЕ-КОМПИЛЯЦИЯ XSLT-ДОКУМЕНТА

В фабрике `TransformerFactory` есть метод `newTemplates()`, который позволяет представить XSLT-документ в компилированной форме, чтобы в дальнейшем его можно было использовать в нескольких потоках для быстрого получения объекта `Transformer`. Трансформер не поддерживает многопоточность, то есть один экземпляр трансформера нельзя использовать в нескольких потоках одновременно, иначе могут возникнуть коллизии, так как доступ к внутреннему состоянию трансформера не синхронизирован. На входе метод `newTemplates()` принимает объект типа Source, представляющий XSLT-документ, а возвращает объект типа `Templates`, являющийся поддерживающей многопоточность тонкой оболочкой над компилированным представлением XSLT-документа. В дальнейшем для получения трансформера нужно вызывать метод `newTransformer()` на объекте `Templates`.

Из объекта `Templates` можно получить параметры вывода (атрибуты элемента `<xsl:output>`) с помощью метода `getOutputProperties()`. Параметры возвращаются в виде объекта `java.util.Properties`, хранящего пары «имя - значение» Полученные параметры можно изменить и передать трансформеру через метод `setOutputProperties()`. Благодаря этому можно переопределить параметры вывода, заданные в XSLT-документе.

ОБРАБОТКА ОШИБОК

С помощью метода `setErrorListener()` к фабрике можно прикрепить обработчик ошибок типа `javax.xml.transform.ErrorListener`, который используется на этапе анализа XSLT-документа, а не на этапе собственно преобразования.

Также обработчик ошибок типа `ErrorListener` можно прикрепить к объекту `Transformer` методом `setErrorListener()`. В этом случае обработчик будет получать сообщения об ошибках при выполнении преобразования.

Интерфейс обработчика ошибок `javax.xml.transform.ErrorListener`, аналогично интерфейсу `org.xml.sax.ErrorHandler`, содержит три метода, вызываемых в зависимости от категории ошибки:
1) `warning()` – для предупреждений;
2) `error()` – для нефатальных ошибок;
3) `fatalError()` – для фатальных ошибок.
Все перечисленные методы принимают в качестве параметра исключение типа `javax.xml.transform.TransformerException`, а также выбрасывают исключения этого типа.

Трансформер или фабрика после обнаружения фатальной ошибки при анализе XSLT-документа или при выполнении преобразования может не продолжать свои действия, после нефатальной ошибки или предупреждения анализ или преобразование обычно продолжается. Чтобы гарантированно прервать выполнение преобразование обработчик ошибки должен выбросить исключение `TransformerException` из метода обработки.

В классе исключения `TransformerException` есть метод `getLocator()`, облегчающий локализацию ошибки. Он возвращает объект типа `SourceLocator`, в котором определены те же методы, что и в `org.xml.sax.Locator` (получение строки, столбца, публичного и системного идентификаторов XML-документа).

XSLT PLUGABILITY

Механизм полностью аналогичен механизму DOM Plugability, только вместо поиска неабстрактного подкласса `DocumentBuilderFactory` выполняется поиск неабстрактного подкласса класса `TransformerFactory`. Поиск также включает в себя 4 этапа:

1) анализ системного свойства `javax.xml.transform.TransformerFactory`

2) поиск свойства с этим именем в файле `jaxp.properties` в подкаталоге `lib` каталога установки JRE

3) просмотр всех доступных виртуальной машине во время выполнения приложения JAR-архивов в поисках файла `META-INF/Services/javax.xml.transform.TransformerFactory`, содержащего имя подкласса, объекты которого фактически создаются

4) используется класс, специфический для данной реализации виртуальной машины.

Если неабстрактный подкласс класса `TranformerFactory` не найден или невозможно создать объект найденного подкласса, то выбрасывается исключение `TranformerFactoryConfigurationError`.

ПРИМЕР ИСПОЛЬЗОВАНИЯ

Пример преобразования, использующего DOM-деревья для представления XSLT-документа, источника и результата преобразования.

```
TransformerFactory tfactory = TransformerFactory.newInstance();
// Убедимся, что фабрика поддерживает DOM feature.
if (tfactory.getFeature(DOMSource.FEATURE) &&
        tfactory.getFeature(DOMResult.FEATURE)) {

    DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
    dfactory.setNamespaceAware(true); // обязательно для XSLT
    DocumentBuilder docBuilder = dfactory.newDocumentBuilder();

    // Создание объекта Templates из DOM-дерева.
    Node xslDOM = docBuilder.parse(xslID);
    DOMSource dsource = new DOMSource(xslDOM, xslID);
    Templates templates = tfactory.newTemplates(dsource);
```

```
    // Создание источника в виде DOM-дерева.
    Node sourceNode = docBuilder.parse(sourceID);

    // Создание пустого результата преобразования DOMResult.
    DOMResult dresult = new DOMResult();

    // Собственно преобразование.
    Transformer transformer = templates.newTransformer();
    transformer.transform(new DOMSource(sourceNode, sourceID), dresult);

    // Получение корня результирующего DOM-дерева.
    Node out = dresult.getNode();

    // Вывод его в системный поток вывода System.out для диагностики.
    Transformer serializer = tfactory.newTransformer();
    serializer.transform(new DOMSource(out), new StreamResult(System.out));
}
```

ПРЕОБРАЗОВАНИЕ, СОВМЕЩЕННОЕ С SAX-АНАЛИЗОМ

Помимо рассмотренного выше подхода к XSLT-преобразованию с использованием класса Transformer, существует подход, основанный на совмещении преобразования с SAX-анализом. При этом преобразование осуществляется особым обработчиком содержимого.

Данный подход реализуется с помощью абстрактного класса `SAXTransformerFactory`, располагающегося в пакете `javax.xml.transform.sax` и расширяющего класс `TransformerFactory`. Чтобы определить, поддерживается ли данный подход используемой реализацией XSLT, нужно запросить у фабрики трансформеров feature с названием `SAXTransformerFactory.FEATURE`. При положительном ответе можно безопасно преобразовать фабрику к типу `SAXTransformerFactory`.

Преобразование, совмещенное с SAX-анализом, выполняется обработчиком содержимого типа `TransformerHandler`, для создания которого предназначен метод `newTransformerHandler()` фабрики `SAXTransformerFactory`. Этому методу передается источник XSLT-документа (объект типа `Source`) или скомпилированное преобразование (объект типа `Templates`).

Приемник результата преобразования устанавливается методом `TransformerHandler.setResult()`. Преобразующий обработчик инкапсулирует объект-трансформер, который можно извлечь методом `getTransformer()`, чтобы установить параметры преобразования и параметры вывода.

Компиляцию преобразования также можно совместить с SAX-анализом, для чего предназначен обработчик содержимого типа `TemplatesHandler`. Создается такой обработчик методом `newTemplatesHandler()` фабрики `SAXTransformerFactory`.

После окончания SAX-анализа скомпилированное преобразование можно получить с помощью метода `getTemplates()` обработчика `TemplatesHandler`.

Следующий пример показывает конструирование шаблона преобразования (объекта `Templates`) в процессе SAX-анализа XSLT-документа с помощью `TemplatesHandler`, а также выполнение XSLT-преобразования в процессе SAX-анализа с помощью `TransformerHandler`.

```
TransformerFactory tfactory = TransformerFactory.newInstance();
// Поддерживает ли фабрика SAX features?
if (tfactory.getFeature(SAXTransformerFactory.FEATURE)) {
    // Если да, то можно безопасно привести тип.
    SAXTransformerFactory stfactory = (SAXTransformerFactory) tfactory;

    TemplatesHandler handler = stfactory.newTemplatesHandler();
    // Если этого не сделать, то TemplatesHandler не будет знать,
    // как получить ресурсы по относительным URL.
    handler.setSystemId(xslID);

    // Создать анализатор и установить TemplatesHandler
    // обработчиком содержимого.
    XMLReader reader = XMLReaderFactory.createXMLReader();
    reader.setContentHandler(handler);

    // Проанализировать исходный XSLT-документ.
    reader.parse(xslID);

    // Получить объект-шаблон Templates от обработчика.
    Templates templates = handler.getTemplates();

    // TransformerHandler – это обработчик содержимого, выполняющий
    // XSLT-преобразование над передаваемыми ему SAX-событиями.
    TransformerHandler handler2 = stfactory.newTransformerHandler(templates);

    // Сериализация результата преобразования в системный вывод System.out.
    handler2.setResult(new StreamResult(System.out));
    handler2.getTransformer().setParameter("a-param", "hello to you!");

    // Изменить обработчик содержимого у анализатора.
    reader.setContentHandler(handler);

    // TransformerHandler расширяет интерфейс LexicalHandler,
    // поэтому анализатор нужно настроить на отправку лексических событий.
    reader.setProperty("http://xml.org/sax/properties/lexical-handler",
                       handler);

    // Проанализировать XML-документ, одновременно выполняя
    // XSLT-преобразование.
    reader.parse(sourceID);
}
```

Интересной возможностью является конвейерное преобразование (следующее XSLT-преобразование выполняется над результатом предыдущего), которое реализуется с помощью SAX-фильтров (объекты типа `org.xml.sax.XMLFilter`). Для создания фильтра, выполняющего XSLT-преобразование, предназначен метод `newXMLFilter()` фабрики `SAXTransformerFactory`. Этому методу передается источник XSLT-документа (объект типа `Source`) или скомпилированное преобразование (объект типа `Templates`).

Чтобы убедиться в поддержке конвейерного преобразования, нужно запросить у фабрики трансформеров feature с названием `SAXTransformerFactory.FEATURE_XMLFILTER`.

Для выстраивания фильтров в цепочку используется метод `XMLFilter.setParent()`, которому передается SAX-анализатор или предшествующий фильтр. Собственно конвейерное преобразование, совмещенное с SAX-анализом, выполняется методом `parse()` последнего фильтра в цепочке.

Следующий пример демонстрирует конвейерное преобразование. Каждый фильтр указывает на родительский анализатор `XMLReader`, а последнее преобразование запускается путем вызова метода `parse()` на последнем анализаторе в цепочке.

```
TransformerFactory tfactory = TransformerFactory.newInstance();
// Убедимся, что фабрика поддерживает SAX-фильтрацию.
if (tfactory.getFeature(SAXTransformerFactory.FEATURE_XMLFILTER)) {
    SAXTransformerFactory stf = (SAXTransformerFactory)tfactory;
    XMLReader reader = XMLReaderFactory.createXMLReader();
    XMLFilter filter1 = stf.newXMLFilter(new StreamSource(xslID_1));
    XMLFilter filter2 = stf.newXMLFilter(new StreamSource(xslID_2));
    XMLFilter filter3 = stf.newXMLFilter(new StreamSource(xslID_3));

    // filter1 будет использовать SAX-анализатор в качестве читателя.
    filter1.setParent(reader);

    // filter2 будет использовать transformer1 в качестве читателя.
    filter2.setParent(filter1);

    // filter3 будет использовать filter2 в качестве читателя.
    filter3.setParent(filter2);
    filter3.setContentHandler(new ExampleContentHandler());

    // При вызове разбора на filter3, он установит себя в качестве
    // ContentHandler для filter2 и вызовет filter2.parse(),
    // который установит себя как обработчик содержимого для filter1
    // и вызовет filter1.parse(),который установит себя как
    // обработчик содержимого для SAX-анализатора reader и вызовет
    // reader.parse(new InputSource("xml/foo.xml")).

    filter3.parse(new InputSource(sourceID));
}
```

Лекция 10.

ПОДДЕРЖКА ВАЛИДАЦИИ XML-ДОКУМЕНТОВ В JAXP

Пакет `javax.xml.validation` содержит средства для работы со схемами и анализа соответствия XML-документа схеме. Прежде валидация по схеме рассматривалась как дополнительная особенность XML-анализатора и анализ соответствия выполнялся самим анализатором. При этом было невозможно обеспечить поддержку нового языка схем, не модифицируя XML-анализатор. В настоящее время поддержки единственного языка XML-схем от W3C недостаточно, так как набирают популярность конкуренты в лице OASIS RELAX NG 1.0 и Schematron (находится в процессе утверждения ISO). Новый механизм позволяет добавить поддержку новых языков схем без изменения XML-анализатора.

ФАБРИКА СХЕМ И СХЕМА

В пакете `javax.xml.validation` есть два основных класса: `SchemaFactory` и `Schema`.
XML-схема представляется объектом класса `Schema`, который создается при помощи абстрактной фабрики `SchemaFactory`. Метод `SchemaFactory.newInstance()` для создания экземпляра фабрики имеет строковый параметр, который указывает язык схем.

Язык схем идентифицируется URI (пространством имен). Для удобства в классе `XMLConstants` определены константы для распространенных языков схем.

Язык схем	URI	Константа в `XMLConstants`
W3C XML Schema	`http://www.w3.org/2001/XMLSchema`	`W3C_XML_SCHEMA_NS_URI`
OASIS RELAX NG 1.0	`http://relaxng.org/ns/structure/1.0`	`RELAXNG_NS_URI`

Поддержка языка схем W3C является обязательной (для JAXP 1.4).
Проверка документа на соответствие DTD не предусмотрена, так как (как отмечается в спецификации) природа DTD такова, что очень трудно организовать соответствующую проверку таким образом, что она была отделена от собственно анализа XML-документа.

Для конфигурирования фабрики поддерживается механизм установки свойств логического и объектного типа (методы `get-/setFeature()` и `get-/setProperty()`, соответственно).
Все реализации должны поддерживать feature, название которой можно посмотреть в `XMLConstants.FEATURE_SECURE_PROCESSING`. Если значение данной feature – `true`, то анализ схемы выполняется с учетом ограничений по памяти и безопасности, вследствие чего не проверяются некоторые конструкции языка XML-схем.

Получив фабрику, можно создать с ее помощью объект типа `Schema`. Для этого предназначены методы `newSchema()`.
Есть версия данного метода без параметров, которая поддерживается только для W3C-схем. В качестве источника документа схемы используется ссылка на схему, внедренная в XML-документ (атрибуты `xsi:schemaLocation` и `xsi:noNamespaceSchemaLocation`). На данном этапе собственно XML-документ неизвестен. Если впоследствии анализируемые документы ссылаются на одну и ту же схему, то она будет проанализирована единожды.
Версии метода с параметрами принимают `File`, `URL`, `javax.xml.transform.Source`, а также массив источников `Source[]`. Последний вариант поддерживается только для W3C-

схем. При этом все источники анализируются и формируют в результате единую схему, эквивалентную пустой схеме, которая импортирует все указанные источники.

Можно обрабатывать ошибки, связанные с анализом самой схемы. Для этого у фабрики есть метод `setErrorHandler()`, который связывает с фабрикой обработчик ошибок – объект типа `org.xml.sax.ErrorHandler`. Правила обработки ошибок такие же, как при обработке документа SAX-анализатором.

VALIDATION PLUGABILITY

Механизм аналогичен механизму DOM Plugability, только вместо поиска неабстрактного подкласса `DocumentBuilderFactory` выполняется поиск неабстрактного подкласса класса `SchemaFactory`. Поиск также включает в себя 4 этапа:

1) анализ системного свойства `javax.xml.validation.SchemaFactory:`*`schemaLanguage`*, где *`schemaLanguage`* – идентификатор языка схем

2) поиск свойства с этим именем в файле `jaxp.properties` в подкаталоге `lib` каталога установки JRE

3) просмотр всех доступных виртуальной машине во время выполнения приложения JAR-архивов в поисках файла `META-INF/Services/javax.xml.validation.SchemaFactory`, содержащего имя подкласса, объекты которого фактически создаются. Поддержка нужного языка схем проверяется у реализации фабрики методом `isSchemaLanguageSupported(`*`schemaLanguage`*`)`.

4) используется класс, специфический для данной реализации виртуальной машины. Любая реализация JAXP должна содержать фабрику, поддерживающую язык схем W3C

Если неабстрактный подкласс класса `SchemaFactory` не найден или невозможно создать объект найденного подкласса, то выбрасывается исключение `IllegalArgumentException`.

ВАЛИДАТОРЫ

Имеется два подхода к валидации – использование независимого валидатора (объекта типа `Validator`), и работа поверх SAX-анализатора, то есть событийно-ориентированная валидация (`ValidatorHandler`).

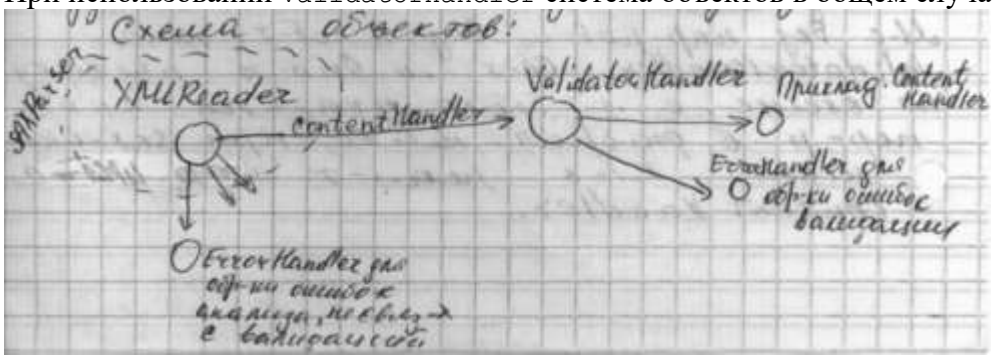
`Schema` – это фабрика для объектов типа `Validator`, которые создаются методом `newValidator()`.
Для конфигурирования валидатора поддерживается механизм установки свойств логического и объектного типа (методы `get-/setFeature()` и `get-/setProperty()`, соответственно).
Метод `reset()` позволяет вернуть валидатор в состояние, в котором он был после создания. Это позволяет повторно использовать объект валидатора, экономя ресурсы и время на создание нового валидатора.
Также можно поменять обработчик ошибок методом `setErrorHandler()`.
Основной метод, выполняющий собственно валидацию – `validate()`. Первая версия принимает единственный параметр типа `Source`, представляющий проверяемый XML-документ. Вторая версия имеет также параметр типа `Result` – XML-документ после

проверки соответствия схеме. После анализа в документе могут быть произведены изменения, в частности, может быть удален игнорируемый пробельный материал, выполнено объединение секций CDATA и текстовых узлов и т.д. Должно быть соответствие между типами источника и результата (DOM- или SAX-источники и результаты, потоковый источник `StreamSource` использовать нельзя).

В классе `Schema` есть метод `newValidatorHandler()` без параметров, возвращающий объект типа `ValidatorHandler`, который может выполнять проверку по схеме в процессе SAX-анализа и реализует интерфейс `ContentHandler`.
Для конфигурирования обработчика валидации поддерживается механизм установки свойств логического и объектного типа (методы `get-/setFeature()` и `get-/setProperty()`, соответственно).
Можно установить обработчик ошибок методом `setErrorHandler()`, который будет обрабатывать ошибки, возникающие в ходе валидации документа по схеме.
Метод `setContentHandler()` позволяет установить объект типа `ContentHandler`, который будет выполнять прикладную обработку XML-документа параллельно с валидацией документа по схеме. `ValidatorHandler` выступает в данном случае своеобразным фильтром.
Последовательность действий:

1. Создаем `SchemaFactory`.
2. Создаем `Schema`.
3. Создаем `ValidatorHandler`.
4. Создаем `SAXParserFactory`.
5. Создаем `SAXParser`.
6. Получаем `XMLReader` из `SAXParser`.
7. Создаем прикладной `ContentHandler`.
8. Прикрепляем прикладной обработчик содержимого к `ValidatorHandler`.
9. Прикрепляем `ValidatorHandler` к `XMLReader`.
10. Вызываем метод `parse()` на объекте `XMLReader`.

На шаге 4 вместо `SAXParserFactory` можно использовать `XMLReaderFactory`, в этом случае шаги 4, 5 и 6 сливаются в один – создание объекта `XMLReader` с помощью фабрики `XMLReaderFactory`.
При использовании `ValidatorHandler` система объектов в общем случае выглядит так:



Также есть возможность прикрепить объект типа `Schema` к фабрике DOM- или SAX-анализаторов (объектам типа `DocumentBuilderFactory` и `SAXParserFactory`, соответственно) с помощью метода `setSchema()`. В этом случае DOM- или SAX-анализаторы, создаваемые данной фабрикой, в процессе анализа будут также выполнять валидацию по схеме. Использование объекта схемы является альтернативой использованию рассмотренных ранее свойств фабрик `http://java.sun.com/xml/jaxp/properties/schemaLanguage` и `http://java.sun.com/xml/jaxp/properties/schemaSource`.

ПРИМЕР ИСПОЛЬЗОВАНИЯ

Следующий пример демонстрирует использование валидации при DOM-анализе. Преимущество подобного использования Validation API: построение DOM-дерева и валидация его по схеме разделены во времени.

```
DocumentBuilder db =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = db.parse(new File("test.xml"));
SchemaFactory sf =
        SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Source schemaFile = new StreamSource(new File("test.xsd"));
Schema schema = sf.newSchema(schemaFile);
Validator validator = schema.newValidator();
try {
    validator.validate(new DOMSource(document));
} catch (SAXException e) { ... }
```

Следующий пример демонстрирует выполняемую при DOM-анализе валидацию XML-документа по схеме.

```
DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
SchemaFactory sf =
        SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Source schemaFile = new StreamSource(new File("test.xsd"));
Schema schema = sf.newSchema(schemaFile);
dbf.setSchema(schema);
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse(new File("test.xml"));
```

Лекция 11. Вычисление XPath-выражений

ПОДДЕРЖКА XPATH В JAXP

Пакет `javax.xml.xpath` содержит программные интерфейсы для вычисления XPath-выражений.

Данный пакет содержит абстрактную фабрику `XPathFactory`, для создания которой предназначен статический метод `XPathFactory.newInstance()`. Фабрика позволяет создавать объекты `XPath` с помощью метода `newXPath()`.

Существует две версии метода `XPathFactory.newInstance()`:

1) с параметром типа `String` – позволяет указать объектную модель, в соответствии с которой будет представлен фрагмент документа, по которому будет вычисляться XPath-выражение;
2) без параметров – используется объектная модель по умолчанию (W3C DOM).

Объектная модель идентифицируется URI. В классе `XPathFactory` определена константа `DEFAULT_OBJECT_MODEL_URI`, которая содержит идентификатор объектной модели, используемой по умолчанию.
По умолчанию используется модель W3C DOM, которая идентифицируется следующим URI: `http://java.sun.com/jaxp/xpath/dom`. Для удобства этот URI представлен также в виде константы `XPathConstants.DOM_OBJECT_MODEL`.

Для конфигурирования фабрики используется механизм feature, поддержка которого осуществляется методами `get-/setFeature()`.
Спецификация требует поддержки feature с именем, задаваемом константой `XMLConstants.FEATURE_SECURE_PROCESSING`. Если значение данной feature – `true`, то использование внешней функции в XPath-выражении считается ошибкой (и должно выбрасываться исключение `XPathFunctionException`).
Если feature не поддерживается, то выбрасывается исключение `XPathFactoryConfigurationException`.

XPATH PLUGABILITY

Механизм аналогичен механизму DOM Plugability, только вместо поиска неабстрактного подкласса `DocumentBuilderFactory` выполняется поиск неабстрактного подкласса класса `XPathFactory`. Поиск также включает в себя 4 этапа:

1) анализ системного свойства `javax.xml.xpath.XPathFactory:`*`uri`*, где *`uri`* – идентификатор объектной модели, по которой вычисляется XPath-выражение

2) поиск свойства с этим именем в файле `jaxp.properties` в подкаталоге `lib` каталога установки JRE

3) просмотр всех доступных виртуальной машине во время выполнения приложения JAR-архивов в поисках файла `META-INF/Services/javax.xml.xpath.XPathFactory`, содержащего имя подкласса, объекты которого фактически создаются. Поддержка нужной объектной модели проверяется у реализации фабрики методом `isObjectModelSupported(`*`uri`*`)`.

4) используется класс, специфический для данной реализации виртуальной машины. Любая реализация должна содержать фабрику, поддерживающую объектную модель W3C DOM

Если неабстрактный подкласс класса `XPathFactory` не найден или невозможно создать объект найденного подкласса, то выбрасывается исключение `XPathFactoryConfigurationException`.

ВЫЧИСЛЕНИЕ XPATH-ВЫРАЖЕНИЙ

Интерфейс `XPath` необходим для компиляции и вычисления XPath-выражений.

Компиляция позволяет снизить накладные расходы при многократном вычислении одного и того же XPath-выражения. Для компиляции используется метод `compile()`, принимающий XPath-выражение в виде строки и возвращающий объект типа `XPathExpression`.

Для непосредственного вычисления XPath-выражения предназначены методы `evaluate()`, которые отличаются типом возвращаемого значения (строка или объект, тип которого задается параметром метода) и типом контекстного узла (произвольный объект – как правило, `Node` или `NodeList`, – либо XML-документ в виде источника `InputSource`).

Тип значения XPath-выражения задается в виде квалифицированного имени (`QName`). Для типов значений, поддерживаемых XPath 1.0, определены константы в классе `XPathConstants` и соответствующие Java-типы (табл. 1).

Константа в классе `XPathConstants`	Тип данных XPath 1.0[1]	Тип Java
`BOOLEAN`	`BOOLEAN`	`java.lang.Boolean`
`NUMBER`	`NUMBER`	`java.lang.Double`
`STRING`	`STRING`	`java.lang.String`
`NODE`	`NODE`	`org.w3c.dom.Node`
`NODESET`	`NODESET`	`org.w3c.dom.NodeList`

Примечания:
1. Указанные типы определены в пространстве имен `http://www.w3.org/1999/XSL/Transform`.

Для вычисления скомпилированного XPath-выражения в классе `XPathExpression` определены методы `evaluate()`. Они полностью аналогичны рассмотренным выше методам класса `XPath`, за исключением параметра, содержащего выражение – для скомпилированного XPath-выражения он не имеет смысла.

Помимо контекстного узла в контекст для вычисления XPath-выражения входят множество переменных, библиотеки функций и множество объявлений пространств имен.

1. В технологии XPath предусмотрена возможность расширения набора доступных функций. Чтобы поддерживать эту возможность в JAXP, в классах `XPathFactory` и `XPath` имеются методы `setXPathFunctionResolver()`, которые принимают объект типа `XPathFunctionResolver`.

Интерфейс `XPathFunctionResolver` содержит единственный метод `resolveFunction()`, вызов которого осуществляется в процессе подготовки к вычислению XPath-выражения. Данный метод имеет два параметра: квалифицированное имя функции и ее арность (количество аргументов функции). Результатом метода является объект типа `XPathFunction`.
Для вычисления значения функции используется метод `evaluate(List)` объекта `XPathFunction`, единственный параметр которого содержит список аргументов функции.
При вычислении функции может быть выброшено исключение `XPathFunctionException`.

2. Для получения значений переменных во время вычисления XPath-выражения используется объект типа `XPathVariableResolver`, который можно установить для фабрики или для объекта `XPath` методом `setXPathVariableResolver()`.
Для получения значения переменной используется метод `resolveVariable()` объекта `XPathVariableResolver`, который принимает квалифицированное имя переменной и возвращает значение переменной в виде произвольного объекта. Если переменной с указанным именем не существует, то возвращается значение `null`.

3. Для управления пространствами имен, доступными при вычислении XPath-выражения, используется объект типа `javax.xml.namespace.NamespaceContext`. Контекст пространств имен можно получить или установить для объекта `XPath` методами `get-/setNamespaceContext()`.

При вычислении выражения может быть выброшено исключение `XPathExpressionException`.

Метод `reset()` возвращает объект `XPath` к исходному состоянию, в котором он был получен от фабрики.

ПРИМЕР ИСПОЛЬЗОВАНИЯ

В примере выполняется DOM-анализ XML-документа, по которому затем вычисляется XPath-выражение.

```
// анализ XML-документа
DocumentBuilder builder =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
org.w3c.dom.Document document = builder.parse(new File("/widgets.xml"));

// вычисление XPath-выражения в контексте всего документа
XPath xpath = XPathFactory.newInstance().newXPath();
String expression = "/widgets/widget[@name='a']/@quantity";
Double quantity =
    (Double) xpath.evaluate(expression, document, XPathConstants.NUMBER);
```

Лекция 12. JAXB

JAXB (JAVA API FOR XML BINDING)

1. Назначение JAXB, сравнение с DOM

JAXB представляет собой набор программных интерфейсов, предназначенных для работы с XML-документами на качественно ином уровне, нежели рассмотренные ранее средства анализа (DOM, SAX, StAX).

В рамках DOM-анализа XML-документ представляется в виде графа объектов (точнее, дерева объектов), но эти объекты не относятся к конкретной предметной области анализируемого документа (например, заказ, клиент и т.п.), а представляют различные виды XML-разметки (элемент, атрибут, текст и т.д.). Последнее затрудняет прикладную обработку XML-документов, ориентированных на хранение данных. С помощью JAXB исходный XML-документ представляется в виде графа объектов, специфичных для предметной области. Также JAXB выполняет обратную операцию – сохранение графа объектов в виде XML-документа. Таким образом, JAXB выполняет двунаправленное преобразование «объекты – XML».

В некоторых случаях целесообразно комбинирование форм представления XML-документа в виде DOM-дерева и в виде графа объектов, специфичных для предметной области. Так, при обработке XML-документов, ориентированных на хранение текста, весь документ можно представить в виде DOM-дерева, а для работы с отдельными фрагментами, обладающими строгой структурой, применять JAXB. Обратная ситуация возникает при обработке XML-документов, ориентированных на хранение данных, некоторые элементы которых допускают смешанное содержимое. В этом случае весь документ рационально представить в виде графа специфичных объектов, а элементы со смешанным содержимым (или с открытой моделью содержимого) – в виде DOM-деревьев.

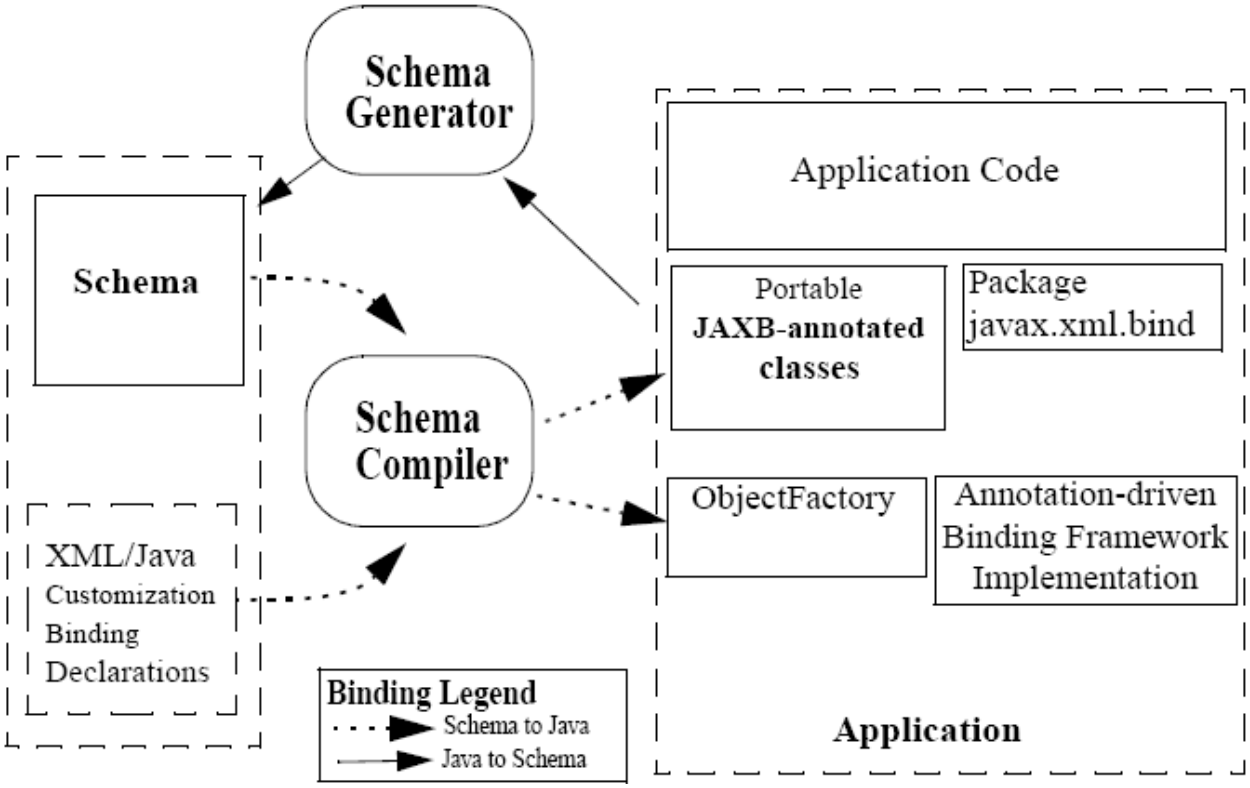
Актуальной версией JAXB является версия 2.0, которая входит в Java SE 6 и Java EE 5. Основное отличие от JAXB 1.0 заключается в переходе от отображения XML-типов на Java-интерфейсы к использованию аннотированных Java-классов. Кроме того, при разработке JAXB 2.0 были приняты во внимание изменения, произошедшие в JAXP 1.4: выделение валидации в отдельный API и добавление StAX-анализа к числу обязательных видов анализа XML-документов.

2. Архитектура JAXB

Архитектура JAXB представлена на рис. 1. Как и традиционные средства объектно-реляционного преобразования, JAXB состоит из двух компонентов (уровней):

1) утилит времени разработки, выполняющих прямое и обратное преобразование между XML-схемой и множеством Java-классов, связанных отношениями ассоциации и наследования;

2) библиотеки времени выполнения, выполняющей преобразование XML-документов в графы объектов и наоборот.

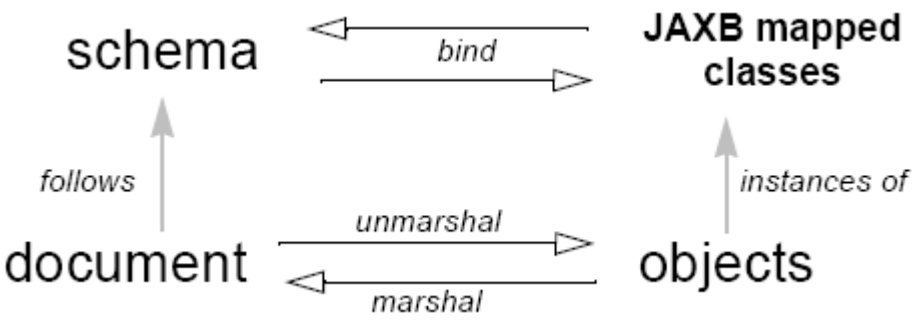


Прямое преобразование между XML-схемой и множеством Java-классов выполняется с помощью компилятора схемы (Schema Compiler). В Sun JDK для вызова компилятора схемы предназначена команда `xjc`. Помимо собственно XML-схемы компилятору также можно указать файл с настройками преобразования (XML/Java Customization Binding Declarations), в противном случае преобразование будет выполнено по правилам, принятым по умолчанию. Компилятор на выходе генерирует: 1) пакет, соответствующий пространству имен XML-схемы; 2) классы, соответствующие типам, объявленным в XML-схеме; 3) фабрику для создания объектов этих классов и глобальных элементов, объявленных в XML-схеме.

Обратное преобразование (множество Java-классов в XML-схему) выполняется с помощью генератора схемы (Schema Generator). В Sun JDK для вызова генератора схемы предназначена команда `schemagen`. Настройки обратного преобразования выполняются в исходных Java-классах при помощи аннотаций. Обратите внимание, что создаваемые компилятором схемы классы также содержат необходимые аннотации, обеспечивая тем самым замкнутый цикл преобразования. Аннотационные типы для JAXB определены в пакете `javax.xml.bind.annotation`.

Библиотека времени выполнения (Binding Framework Implementation) выполняет свою работу на основании аннотаций. Код приложения обращается к функциям библиотеки через классы и интерфейсы, определенные в пакете `javax.xml.bind`. Для прикладной обработки XML-документа в коде приложения используются аннотированные классы. Для создания объектов при преобразовании XML-документа в граф объектов библиотека времени выполнения использует фабрику. Также фабрика объектов используется кодом приложения для создания нового графа объектов или добавления новых объектов в существующий граф.

Связь между понятиями, используемыми во время разработки и во время выполнения, отображена на рис. 2.



3. Контекст

Точкой входа в JAXB для приложения является контекст – объект типа `JAXBContext`, который создается статическими методами `newInstance()`. Контекст связан с множеством классов, соответствующих типам из XML-схемы. При выполнении преобразования могут использоваться только классы, входящие в контекст. Естественно, контекст не должен содержать классов, которые могут вызвать неоднозначность преобразования. Список доступных в контексте классов передается в метод `newInstance()` одним из двух способов: 1) как последовательность объектов типа `Class`; 2) в виде строки, содержащей перечисленные через точку с запятой полноквалифицированные имена классов и пакетов. В первом случае в контекст будут добавлены не только классы, переданные непосредственно, но также и те классы, на которые есть ссылки в переданных классах (например, поля соответствующих типов). Если при инициализации контекста передается имя пакета, то данный пакет должен содержать класс фабрики `ObjectFactory` и/или файл `jaxb.index` со списком отображаемых классов. Таким образом, при обратном преобразовании (Java -> XML) во время разработки не требуется вручную создавать класс фабрики, но может потребоваться дополнительный файл.

4. Демаршалинг

Прямое преобразование (XML -> Java) во время выполнения в JAXB называется демаршалингом и выполняется объектом типа `Unmarshaller`. Для получения последнего предназначен метод контекста `JAXBContext.createUnmarshaller()`.

В интерфейсе `Unmarshaller` определено несколько версий метода `unmarshal()`, выполняющих демаршалинг XML-документа из следующих источников:

1) байтовый поток ввода (`java.io.InputStream`);
2) символьный поток ввода (`java.io.Reader`);
3) файл (`java.io.File`);
4) URL (`java.net.URL`);
5) SAX-совместимый источник ввода (`org.xml.sax.InputSource`);
6) источник преобразования (`javax.xml.transform.Source`);
7) DOM-дерево (`org.w3c.dom.Node`);
8) читатель StAX-событий (`javax.xml.stream.XMLStreamReader` или `javax.xml.stream.XMLEventReader`).

Метод `unmarshal()` возвращает объект типа `JAXBElement<`*`ТипКорневогоЭлемента`*`>` (если передать в качестве параметра *`ТипКорневогоЭлемента`*`.class`) либо объект типа `Object`, который необходимо явно преобразовать к нужному типу. Другими словами, в результате демаршалинга корневой[1] элемент XML-документа возвращается в виде Java-объекта.

Параметризованный класс `JAXBElement<`*`ТипЭлемента`*`>` представляет элемент XML-документа. Из объекта этого класса можно получить имя элемента, тип согласно XML-схеме и некоторые другие характеристики. Значение элемента представляет собой объект

[1] В случае источников типов 6-8 демаршалинг может вестись для части XML-документа, в этом случае возвращается некорневой элемент.

Java-класса, соответствующего типу элемента согласно XML-схеме. Для работы со значением элемента предназначены методы `getValue()` и `setValue(`*`ТипЭлемента`*`)`.

5. Маршалинг

Обратное преобразование (Java -> XML) во время выполнения в JAXB называется маршалингом и выполняется объектом типа `Marshaller`, для получения которого предназначен метод контекста `JAXBContext.createMarshaller()`.

В интерфейсе `Marshaller` определено несколько версий метода `marshal()`, выполняющих маршалинг XML-документа в следующие приемники:

1) байтовый поток вывода (`java.io.OutputStream`);
2) символьный поток вывода (`java.io.Writer`);
3) обработчик содержимого (`org.xml.sax.ContentHandler`);
4) приемник преобразования (`javax.xml.transform.Result`);
5) DOM-дерево (`org.w3c.dom.Node`);
6) писатель StAX-событий (`javax.xml.stream.XMLStreamWriter` или `javax.xml.stream.XMLEventWriter`).

Помимо приемника XML-документа в метод `marshal()` передается корневой элемент документа в виде объекта `JAXBElement<`*`ТипКорневогоЭлемента`*`>`.

В случае программного создания графа объектов с его последующим маршалингом для создания корневого элемента документа можно использовать метод `create`*`ИмяКорневогоЭлемента`*`(`*`ТипКорневогоЭлемента`*`)` фабрики `ObjectFactory`.

6. Валидация и обработка ошибок.

В JAXB поддерживаются следующие виды валидации:

1) валидация при демаршалинге;
2) валидация при маршалинге;
3) валидация «на лету» – выполняется при изменении связанных с XML объектов через `set`-методы. Поддержка данного вида валидации необязательна.

Первые два вида валидации осуществляются по схеме XML-документа средствами Validation API. По умолчанию валидация при маршалинге/демаршалинге не выполняется. Для ее включения необходимо установить для объекта типа `Marshaller`/`Unmarshaller` схему методом `setSchema(Schema)`.

Для обработки ошибок валидации необходимо предоставить обработчик типа `ValidationEventHandler`, который устанавливается для объекта типа `Marshaller`/`Unmarshaller` методом `setEventHandler()`.

В интерфейсе `ValidationEventHandler` определен единственный метод `handleEvent()`, принимающий в качестве параметра событие валидации `ValidationEvent`. Данный метод возвращает логическое значение, которое определяет, продолжить (`true`) или остановить (`false`) текущий процесс маршалинга/демаршалинга. При останове библиотека времени выполнения выбрасывает соответствующее исключение типа `UnmarshalException`.

Событие валидации `ValidationEvent` является оберткой над объектом исключения, возникшего в процессе валидации XML-документа. Метод `getSeverity()` возвращает степень критичности события в виде одной из констант, определенных в интерфейсе `ValidationEvent`: `WARNING` – предупреждение; `ERROR` – нефатальная ошибка; `FATAL_ERROR` – фатальная ошибка. Получить текстовое сообщение о событии можно методом `getMessage()`, а само исключение – методом `getLinkedException()`.

В интерфейсе `ValidationEvent` есть метод `getLocator()`, возвращающий объект типа `ValidationEventLocator`. Данный локатор отличается от рассмотренных ранее (`org.xml.sax.Locator`, `javax.xml.transform.SourceLocator`), так как при валидации «на лету» отсутствует информация об XML-документе. Как следствие, в этом локаторе используется один из трех способов указания местоположения ошибки:

1) URL, смещение в потоке, строка, столбец – при маршалинге/демаршалинге;
2) узел DOM-дерева – при связывании графа объектов с DOM-деревом (т.н. binding),
3) произвольный объект – при валидации «на лету».
Для упрощения обработки ошибок валидации имеется вспомогательный класс `javax.xml.bind.util.ValidationEventCollector`, выполняющий сбор всех событий валидации при маршалинге/демаршалинге. При его использовании следует иметь в виду, что в случае возникновения фатальной ошибки все равно выбрасывается исключение.

7. Отображение XML-схемы на Java-классы по умолчанию
В табл. 1 приведены основные правила, действующие при отображении XML-схемы на Java-классы по умолчанию.

XML-схема	Java
Идентификатор пространства имен	Пакет
Именованный сложный тип	Класс (value class)
Именованный простой тип, ограничивающий тип `xsd:string` фасетами `enum`	Перечислимый тип (`enum`)
Именованный простой тип	Примитивный тип или класс согласно спецификации JAXB
Глобальный элемент именованного типа	Фабрика элемента
Глобальный элемент анонимного типа	Класс для анонимного типа, фабрика типа, фабрика элемента
Атрибут	Свойство (атрибут + `get`/`set`-методы)
Локальный элемент (в т.ч. ссылка на элемент), для которого `maxOccurs == 1`	Свойство
Локальный элемент (в т.ч. ссылка на элемент), для которого `maxOccurs > 1`	Свойство типа `java.util.List`

8. Связь с DOM-деревом
Существует возможность отображения на граф объектов только необходимой части XML-документа, представленного в виде DOM-дерева. Для этого создается специальный связующий объект (binder), который поддерживает связь между представлениями XML-документа в виде объектов JAXB и в виде DOM-дерева и обеспечивает их синхронизацию.
Связующий объект типа `Binder` создается с помощью метода `createBinder()` контекста `JAXBContext`.
Связующий объект содержит методы `unmarshal()` для демаршалинга графа объектов из DOM-дерева и `marshal()` для маршалинга графа объектов JAXB в DOM-дерево, в процессе выполнения которых также устанавливаются связи между объектами соответствующих представлений XML-документа.
Если для связующего объекта установить схему методом `setSchema()`, то при маршалинге/демаршалинге будет выполняться также валидация. Обработка ошибок валидации выполняется по принципам, описанным выше.
Для получения объекта JAXB по узлу DOM-дерева и наоборот предназначены методы `getJAXBNode()` и `getXMLNode()`, соответственно.
Синхронизация представлений XML-документа выполняется методами `updateXML()` и `updateJAXB()`. Их действие аналогично маршалингу и демаршалингу, соответственно. Различие состоит в том, что при обновлении DOM-дерева по объекту JAXB в дереве сохраняются комментарии, инструкции обработки и прочие части XML-документа, которые не могут иметь отображения в JAXB (а также те элементы и атрибуты, для которых отображение не определено). А при обновлении графа объектов JAXB по DOM-

дереву по возможности повторно используются существующие объекты, а не создаются новые.

9. Конвейерная обработка и демаршалинг

Демаршалинг средствами JAXB может выполняться в процессе конвейерной обработки XML-документа. Для встраивания в конвейер существует реализация объекта-демаршалера в виде SAX-обработчика содержимого (объекта типа `ContentHandler`) – интерфейс `UnmarshallerHandler`. Так как данный интерфейс не содержит средств для назначения дочернего обработчика содержимого, то демаршалинг может быть только последней обработкой в конвейере.

Для получения демаршалирующего обработчика предназначен метод `getUnmarshallerHandler()` в интерфейсе `Unmarshaller`, то есть он создается из уже существующего объекта-демаршалера.

После SAX-анализа результирующий граф объектов JAXB можно получить у демаршалирующего обработчика методом `getResult()`.

10. Пример использования

Программа должна преобразовывать входной XML-документ, содержащий сведения о заказе, в HTML-документ, в котором список заказываемых товаров должен быть представлен в виде таблицы.

Входной XML-документ должен соответствовать следующей схеме (см. лекция 2):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://isim.vlsu.ru/schema/order"
            xmlns:tns="http://isim.vlsu.ru/schema/order"
            elementFormDefault="qualified">

   <xsd:complexType name="orderType">
      <xsd:sequence>
         <xsd:element name="customer" type="tns:customerType"/>
         <xsd:element name="lineItems" type="tns:lineItemsType"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="order" type="tns:orderType"/>
   <xsd:complexType name="customerType">
      <xsd:sequence>
         <xsd:element name="firstName" type="xsd:string"/>
         <xsd:element name="lastName" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="lineItemsType">
      <xsd:sequence>
         <xsd:element name="lineItem" type="tns:lineItemType"
             maxOccurs="20"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="lineItemType">
      <xsd:sequence/>
        <xsd:attribute name="sku" type="xsd:positiveInteger"/>
        <xsd:attribute name="quantity" type="xsd:double"/>
    </xsd:complexType>
</xsd:schema>
```

1. В результате выполнения компилятора схемы `xjc` создается структура вложенных каталогов `ru/vlsu/isim/schema/order` (что соответствует пространству имен схемы), а в каталоге `order` генерируются следующие файлы:

1) `package-info.java` – аннотации для всего пакета в целом;
2) `ObjectFactory.java` – фабрика объектов JAXB;
3) `CustomerType.java`, `LineItemsType.java`, `LineItemType.java`, `OrderType.java` – классы для именованных типов, определенных в схеме.

Рассмотрим некоторые из этих файлов подробнее. В Java SE 5 появилась возможность предоставлять дополнительные сведения о пакете Java-классов в виде аннотаций в специальном файле `package-info.java`. Компилятор схемы размещает в файле `package-info.java` сведения о пространстве имен схемы, из которой был сгенерированы классы пакета:

```
@javax.xml.bind.annotation.XmlSchema(
    namespace = "http://isim.vlsu.ru/schema/order",
    elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package ru.vlsu.isim.schema.order;
```

Файл `ObjectFactory.java` содержит определение открытого класса `ObjectFactory`, который содержит фабричные методы для создания глобальных элементов схемы и объектов определенных в схеме типов:

```
@XmlRegistry
public class ObjectFactory {

    private final static QName _Order_QNAME =
        new QName("http://isim.vlsu.ru/schema/order", "order");

    public ObjectFactory() { }

    public LineItemsType createLineItemsType() {return new LineItemsType();}

    public OrderType createOrderType() {return new OrderType();}

    public CustomerType createCustomerType() {return new CustomerType();}

    public LineItemType createLineItemType() {return new LineItemType();}

    @XmlElementDecl(namespace = "http://isim.vlsu.ru/schema/order",
                    name = "order")
    public JAXBElement<OrderType> createOrder(OrderType value) {
        return new JAXBElement<OrderType>(_Order_QNAME,
                                          OrderType.class, null, value);
    }
}
```

Файл `OrderType.java` содержит определение открытого класса `OrderType`, который соответствует типу `orderType`, объявленному в XML-схеме. Аннотация `@XmlAccessorType` определяет тип доступа библиотеки времени выполнения JAXB к состоянию объекта – непосредственно через поля или через `get`/`set`-методы (последнее позволяет программно реализовать дополнительные проверки данных). Этот доступ выполняется библиотекой при маршалинге и демаршалинге. Аннотация `@XmlType` задает соответствие между Java-классом и типом XML-схемы, а аннотация `@XmlElement` – между свойством Java-класса и локальным элементом XML-схемы.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "orderType", propOrder = {
    "customer",
    "lineItems"
})
```

```
public class OrderType {

    @XmlElement(required = true)
    protected CustomerType customer;
    @XmlElement(required = true)
    protected LineItemsType lineItems;

    public CustomerType getCustomer() { return customer; }

    public void setCustomer(CustomerType value) { this.customer = value; }

    public LineItemsType getLineItems() { return lineItems; }

    public void setLineItems(LineItemsType value) { this.lineItems = value; }
}
```

2. Поставленная задача решается в классе Main, который выполняет демаршалинг исходного XML-документа с валидацией по схеме. Результирующий HTML-документ выводится в стандартный вывод, а его формирование выполняется в процессе обхода графа объектов JAXB:

```
public class Main {
    public static void main(String[] args) {
        try {
            JAXBContext context =
                JAXBContext.newInstance("ru.vlsu.isim.schema.order");
            Unmarshaller unmarshaller = context.createUnmarshaller();

            SchemaFactory sf =
               SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema schema = sf.newSchema(new File("order.xsd"));
            unmarshaller.setSchema(schema);

            JAXBElement<OrderType> orderElement = null;
            orderElement = (JAXBElement<OrderType>) unmarshaller.unmarshal(
                                new File("orderSample.xml"));

            // прикладной анализ
            OrderType order = orderElement.getValue();
            CustomerType customer = order.getCustomer();
            System.out.println("<html><head><title>Заказ</title></head>");
            System.out.println("<body><h2>Заказ</h2>");
            System.out.println("<body><h3>" + customer.getFirstName()
                + " " + customer.getLastName() + "</h3>");
            System.out.println("<table><thead><tr>"
                + "<th>Товар</th><th>Количество</th></tr></thead><tbody>");

            for (LineItemType lineItem : order.getLineItems().getLineItem())
                System.out.println("<tr><td>" + lineItem.getSku()
                    + "</td><td>" + lineItem.getQuantity() + "</td></tr>");

            System.out.println("</tbody></table></body></html>");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```