Глава 16
XML & JAVA

XML (*Extensible Markup Language* – расширяемый язык разметки) – рекомендован W3C как язык разметки, представляющий свод общих синтаксических правил. XML предназначен для обмена структурированной информацией с внешними системами. Формат для хранения должен быть эффективным, оптимальным с точки зрения потребляемых ресурсов (памяти, и др.). Такой формат должен позволять быстро извлекать полностью или частично хранимые в этом формате данные и быстро производить базовые операции над этими данными.

XML является упрощённым подмножеством языка SGML. На основе XML разрабатываются более специализированные стандарты обмена информацией (общие или в рамках организации, проекта), например XHTML, SOAP, RSS, MathML.

Основная идея XML – это текстовое представление с помощью тегов, структурированных в виде дерева данных. Древовидная структура хорошо описывает бизнес-объекты, конфигурацию, структуры данных и т.п. Данные в таком формате легко могут быть как построены, так и разобраны на любой системе с использованием любой технологии – для этого нужно лишь уметь работать с текстовыми документами. С другой стороны, механизм **namespace**, различная интерпретация структуры XML документа (триплеты RDF, microformat) и существование смешанного содержания (mixed content) часто превращают XML в многослойную структуру, в которой отсутствует древовидная организация (разве что на уровне синтаксиса).

Почти все современные технологии стандартно поддерживают работу с XML. Кроме того, такое представление данных удобочитаемо (human-readable). Если нужен тег для представления имени, его можно создать:

<name>Java SE 6</name> или **<name/>**.

Далее представлены примеры неправильных написаний тегов:

```
<?xml version="1.0"?>
<book>
   <title>title</title>
</book>
<book/>
```

Каждый XML-документ должен содержать только один корневой элемент (root element или document element). В примере есть два корневых элемента, один из которых пустой. В отличие от файла XML, файл HTML может иметь несколько корневых элементов и не обязательно <HTML>.

```
<?xml version="1.0"?>
<book>
   <caption>C++
</book>
   </caption>
```

Тег должен закрываться в том же теге, в котором был открыт. В данном случае это **caption**. В HTML этого правила не существует.

```
<?xml version="1.0"?>
<book>
   <author>Petrov
</book>
```

Любой открывающий тег должен иметь закрывающий. Если тег не имеет содержимого, можно использовать конструкцию вида **<author/>**. В HTML есть возможность не закрывать теги, и браузер определяет стили по открывающемуся тегу

Наименования тегов являются чувствительные к регистру (case-sensitive), т.е. например теги, **<author>**, **<Author>**, **<AuToR>** будут совершенно разными при работе с XML:

```
<author>Petrov</Author>
```

Программа-анализатор просто не найдет завершающий тег и выдаст ошибку. Язык HTML нетребователен к регистру.

Все атрибуты тегов должны быть заключены либо в одинарные, либо в двойные кавычки:

```
<book dateOfIssue="09/09/2007" title='JAVA in Belarus'/>
```

В HTML разрешено записывать значение атрибута без кавычек.

Например: **<FORM method=POST action=index.jsp>**

Пусть существует XML-документ с данными о студентах:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE students SYSTEM "students.dtd">
<students>
   <student login="mit" faculty="mmf">
      <name>Mitar Alex</name>
      <telephone>2456474</telephone>
      <address>
         <country>Belarus</country>
         <city>Minsk</city>
         <street>Kalinovsky 45</street>
      </address>
   </student>
   <student login="pus" faculty="mmf">
      <name>Pashkun Alex</name>
      <telephone>3453789</telephone>
      <address>
         <country>Belarus</country>
         <city>Brest</city>
         <street>Knorina 56</street>
      </address>
   </student>
</students>
```

Каждый документ начинается декларацией – строкой, указывающей как минимум версию стандарта XML. В качестве других атрибутов могут быть указаны кодировка символов и внешние связи.

После декларации в XML-документе могут располагаться ссылки на документы, определяющие структуру текущего документа и собственно XML-элементы (теги), которые могут иметь атрибуты и содержимое. Открывающий тег состоит из имени элемента, например **<city>**. Закрывающий тег состоит из того же имени, но перед именем добавляется символ '**/**', например **</city>**. Содержимым элемента (content) называется всё, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы.

Инструкции по обработке

XML-документ может содержать инструкции по обработке, которые используются для передачи информации в работающее с ним приложение. Инструкция по обработке может содержать любые символы, находиться в любом месте XML документа и должна быть заключены между **<?** и **?>** и начинаться с идентификатора, называемого **target** (цель).

Например:

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

Эта инструкция по обработке сообщает браузеру, что для данного документа необходимо применить стилевую таблицу (stylesheet) **book.xsl**.

Комментарии

Для написания комментариев в XML следует заключать их, как и в HTML, между **<!--** и **-->** . Комментарии можно размещать в любом месте документа, но не внутри других комментариев:

```
<!-- комментарий <!-- Неправильный -->  -->
```

Внутри значений атрибутов:

```
<book title="BLR<!-- Неправильный комментарий -->"/>
```

Внутри тегов:

```
<book <!-- Неправильный комментарий -->/>
```

Указатели

Текстовые блоки XML-документа не могут содержать символов, которые служат в написании самого XML: **<**, **>**, **&**.

```
<description>
      в текстовых блоках нельзя использовать символы <,>,&
</description>
```

В таких случаях используются ссылки (указатели) на символы, которые должны быть заключены между символами **&** и **;** .

Особо распространенными указателями являются:

< – символ **<**;

> – символ **>**;

& – символ **&**;

' – символ апострофа **'**;

" – символ двойной кавычки **"**.

Таким образом, пример правильно будет выглядеть так:

```
<description>
      в текстовых блоках нельзя использовать символы
      &lt;, &gt;, &amp;
</description>
```

Раздел CDATA

Если необходимо включить в XML-документ данные (в качестве содержимого элемента), которые содержат символы '**<**', '**>**', '**&**', ' ' ' и ' " ', чтобы не заменять их на соответствующие определения, можно все эти данные включить в раздел **CDATA**. Раздел **CDATA** начинается со строки "**<[CDATA[**", а заканчивается "**]]>**", при этом между ними эти строки не должны употребляться. Объявить раздел **CDATA** можно, например, так:

```
<data><[CDATA[ 5 < 7 ]]></data>
```

Корректность XML-документа определяют следующие два компонента:

- синтаксическая корректность (well-formed): то есть соблюдение всех синтаксических правил XML;
- действительность (valid): то есть данные соответствуют некоторому набору правил, определённых пользователем; правила определяют структуру и формат данных в XML. Валидность XML документа определяется наличием DTD или XML-схемы XSD и соблюдением правил, которые там приведены.

DTD

Для описания структуры XML-документа используется язык описания DTD (Document Type Definition). В настоящее время DTD практически не используется и повсеместно замещается XSD. DTD может встречаться в достаточно старых приложениях, использующих XML и, как правило, требующих нововведений (upgrade).

DTD определяет, какие теги (элементы) могут использоваться в XML-документе, как эти элементы связаны между собой (например, указывать на то, что элемент **<student>** включает дочерние элементы **<name>**, **<telephone>** и **<address>**), какие атрибуты имеет тот или иной элемент.

Это позволяет наложить четкие ограничения на совокупность используемых элементов, их структуру, вложенность.

Наличие DTD для XML-документа не является обязательным, поскольку возможна обработка XML и без наличия DTD, однако в этом случае отсутствует средство контроля действительности (validness) XML-документа, то есть правильности построения его структуры.

Чтобы сформировать DTD, можно создать либо отдельный файл и описать в нем структуру документа, либо включить DTD-описание непосредственно в документ XML.

В первом случае в документ XML помещается ссылка на файл DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<! DOCTYPE students SYSTEM "students.dtd">
```

Во втором случае описание элемента помещается в XML-документ:

```
<?xml version="1.0" ?>
<! DOCTYPE student [
<!ELEMENT student (name, telephone, address)>
<!--
далее идет описание элементов name, telephone, address
-->
]>
```

Описание элемента

Элемент в DTD описывается с помощью дескриптора **!ELEMENT**, в котором указывается название элемента и его содержимое. Так, если нужно определить элемент **<student>**, у которого есть дочерние элементы **<name>**, **<telephone>** и **<address>**, можно сделать это следующим образом:

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
```

В данном случае были определены три элемента: **name, telephone** и **address** и описано их содержимое с помощью маркера **PCDATA**. Это говорит о том, что элементы могут содержать любую информацию, с которой может работать программа-анализатор (**PCDATA** – parsed character data). Есть также маркеры **EMPTY** – элемент пуст и **ANY** – содержимое специально не описывается.

При описании элемента **<student>**, было указано, что он состоит из дочерних элементов **<name>**, **<telephone>** и **<address>**. Можно расширить это описание с помощью символов '**+**'(один или много), '*****'(0 или много), '**?**'(0 или 1), используемых для указания количества вхождений элементов. Так, например,

```
<!ELEMENT student (name, telephone, address)>
```

означает, что элемент **student** содержит один и только один элемент **name**, **telephone** и **address**. Если существует несколько вариантов содержимого элементов, то используется символ '|' (или). Например:

```
<!ELEMENT student (#PCDATA | body)>
```

В данном случае элемент **student** может содержать либо дочерний элемент **body**, либо **PCDATA**.

Описание атрибутов

Атрибуты элементов описываются с помощью дескриптора **!ATTLIST**, внутри которого задаются имя атрибута, тип значения, дополнительные параметры и имеется следующий синтаксис:

```
<!ATTLIST название_елемента название_атрибута тип_атрибута
значение_по_умолчанию >
```

Например:

```
<!ATTLIST student
   login ID #REQUIRED
   faculty CDATA #REQUIRED>
```

В данном случае у элемента **<student>** определяются два атрибута: **login**, **faculty**. Существует несколько возможных значений атрибута, это:

CDATA – значением атрибута является любая последовательность символов;

ID – определяет уникальный идентификатор элемента в документе;

IDREF (**IDREFS**) – значением атрибута будет идентификатор (список идентификаторов), определенный в документе;

ENTITY (**ENTITIES**) – содержит имя внешней сущности (несколько имен, разделенных запятыми);

NMTOKEN (**NMTOKENS**) – слово (несколько слов, разделенных пробелами).

Опционально можно задать значение по умолчанию для каждого атрибута. Значения по умолчанию могут быть следующими:

#REQUIRED – означает, что атрибут должен присутствовать в элементе;

#IMPLIED – означает, что атрибут может отсутствовать, и если указано значение по умолчанию, то анализатор подставит его.

#FIXED – означает, что атрибут может принимать лишь одно значение, то, которое указано в DTD.

defaultValue – значение по умолчанию, устанавливаемое парсером при отсутствии атрибута. Если атрибут имеет параметр **#FIXED**, то за ним должно следовать **defaultValue**.

Если в документе атрибуту не будет присвоено никакого значения, то его значение будет равно заданному в DTD. Значение атрибута всегда должно указываться в кавычках.

Определение сущности

Сущность (entity) представляет собой некоторое определение, чье содержимое может быть повторно использовано в документе. Описывается сущность с помощью дескриптора **!ENTITY**:

```
<!ENTITY company 'Sun Microsystems'>
<sender>&company;</sender>
```

Программа-анализатор, которая будет обрабатывать файл, автоматически подставит значение `Sun Microsystems` вместо **&company**.

Для повторного использования содержимого внутри описания DTD используются параметрические (параметризованные) сущности.

```
   <!ENTITY   % elementGroup "firstName, lastName,gender,
address, phone">
   <!ELEMENT employee (%elementGroup;)>
   <!ELEMENT contact (%elementGroup)>
```

В XML включены внутренние определения для символов. Кроме этого, есть внешние определения, которые позволяют включать содержимое внешнего файла:

```
<!ENTITY logotype SYSTEM "/image.gif" NDATA GIF87A>
```

Файл DTD для документа **students.xml** будет иметь вид:

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT students (student)+>
<!ELEMENT student (name, telephone, address)>
<!ATTLIST student
   login ID #REQUIRED
   faculty CDATA #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
```

Схема XSD

Схема XSD представляет собой более строгое, чем DTD, описание XML-документа. XSD-схема, в отличие от DTD, сама является XML-документом и поэтому более гибкая для использования в приложениях, задания правил документа, дальнейшего расширения новой функциональностью. В отличиеи от DTD, эта схема содержит много базовых типов (44 типа) и имеет поддержку пространств имен (namespace). С помощью схемы XSD можно также проверить документ на корректность.

Схема XSD первой строкой должна содержать декларацию XML

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

Любая схема своим корневым элементом должна содержать элемент **schema**.

Для создания схемы нужно описать все элементы: их тип, количество повторений, дочерние элементы. Сам элемент создается элементом **element**, который может включать следующие атрибуты:

ref – ссылается на определение элемента, находящегося в другом месте;

name – определяет имя элемента;

type – указывает тип элемента;

minOccurs и **maxOccurs** – количество повторений этого элемента (по умолчанию **1**), чтобы указать, что количество элементов неограниченно, в атрибуте **maxOccurs** нужно задать **unbounded**.

Если стандартные типы не подходят, можно создать свой собственный тип элемента. Типы элементов делятся на простые и сложные. Различия заключаются в том, что сложные типы могут содержать другие элементы, а простые – нет.

Простые типы

Элементы, которые не имеют атрибутов и дочерних элементов, называются простыми и должны иметь простой тип данных.

Существуют стандартные простые типы, например **string** (представляет строковое значение), **boolean** (логическое значение), **integer** (целое значение), **float** (значение с плавающей точкой), **ID** (идентификатор) и др. Также простые типы можно создавать на основе существующих типов посредством элемента **simpleType**. Атрибут **name** содержит имя типа.

Все типы в схеме могут быть объявлены как локально внутри элемента, так и глобально с использованием атрибута **name** для ссылки на тип в любом месте схемы. Для указания основного типа используется элемент **restriction**. Его атрибут **base** указывает основной тип. В элемент **restriction** можно включить ряд ограничений на значения типа:

minInclusive – определяет минимальное число, которое может быть значением этого типа;

maxInclusive – максимальное значение типа;

length – длина значения;

pattern – определяет шаблон значения;

enumeration – служит для создания перечисления.

Следующий пример описывает тип **Login**, производный от **ID** и отвечающий заданному шаблону в элементе **pattern**.

```
<simpleType name="Login">
   <restriction base="ID">
      <pattern value="[a-zA-Z]{3}[a-zA-Z0-9_]+"/>
   </restriction>
</simpleType>
```

Сложные типы

Элементы, содержащие в себе атрибуты и/или дочерние элементы, называются сложными.

Сложные элементы создаются с помощью элемента **complexType**. Так же как и в простом типе, атрибут **name** задает имя типа. Для указания, что элементы должны располагаться в определенной последовательности, используется элемент **sequence**. Он может содержать элементы **element**, определяющие содержание сложного типа. Если тип может содержать не только элементы, но и текстовую информацию, необходимо задать значение атрибута **mixed** в **true**. Кроме элементов, тип может содержать атрибуты, которые создаются элементом **attribute**. Атрибуты элемента **attribute**: **name** – имя атрибута, **type** – тип значения атрибута. Для указания, обязан ли использоваться атрибут, нужно использовать атрибут **use**, который принимает значения **required**, **optional**, **prohibited**. Для установки значения по умолчанию используется атрибут **default** , а для фиксированного значения – атрибут **fixed**.

Следующий пример демонстрирует описание типа **Student**:

```
<complexType name="Student">
   <sequence>
      <element name="name" type="string"/>
      <element name="telephone" type="decimal"/>
      <element name="address" type="tns:Address"/>
   </sequence>
   <attribute name="login" type="tns:Login"
      use="required"/>
   <attribute name="faculty" type="string"
      use="required"/>
</complexType>
```

Для объявления атрибутов в элементах, которые могут содержать только текст, используются элемент **simpleContent** и элемент **extension**, с помощью которого расширяется базовый тип элемента атрибутом(ами).

```
<element name="Student">
  <complexType>
   <simpleContent>
      <extension base="string">
         <attribute name="birthday" type="string"/>
      </extension>
   </simpleContent>
  </complexType>
</element>
```

Для расширения/ограничения ранее объявленных сложных типов используется элемент **complexContent**.

```
<complexType name="personType">
      <sequence>
            <element name="firstName" type="string"/>
            <element name="lastName" type="string"/>
            <element name="address" type="string"/>
      </sequence>
</complexType>
<complexType name="studentType">
   <complexContent>
      <extension base="personType">
            <sequence>
                  <element name="course" type="integer"/>
                  <element name="faculty" type="string"/>
            </sequence>
      </extesion>
   </complexContent>
</complexType>
<element name="Student" type="studentType"/>
```

Для задания порядка следования элементов в XML используются такие теги, как **<all>**, который допускает любой порядок.

```
<element name="person">
      <complexType>
            <all>
                  <element name="firstName" type="string"/>
                  <element name="lastName" type="string"/>
            </all>
      </complexType>
</element>
```

Элемент **<choice>** указывает, что в XML может присутствовать только один из перечисленных элементов. Элемент **<sequence>** задает строгий порядок дочерних элементов.

Для списка студентов XML-схема **students.xsd** может выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.example.com/Students"
      xmlns:tns="http://www.example.com/Students">
      <element name="students">
            <complexType>
                  <sequence>
                        <element name="student"
type="tns:Student" minOccurs="1" maxOccurs="unbounded" />
                  </sequence>
            </complexType>
      </element>

      <complexType name="Student">
            <sequence>
```

```
            <element name="name" type="string" />
            <element name="telephone" type="decimal" />
            <element name="address" type="tns:Address" />
          </sequence>
          <attribute name="login" type="tns:Login"
use="required" />
          <attribute name="faculty" type="string"
use="required" />
    </complexType>

    <simpleType name="Login">
          <restriction base="ID">
             <pattern value="[a-zA-Z]{3}[a-zA-Z0-9_]*"/>
          </restriction>
    </simpleType>

    <complexType name="Address">
          <sequence>
                <element name="country" type="string" />
                <element name="city" type="string" />
                <element name="street" type="string" />
          </sequence>
    </complexType>
</schema>
```

В приведенном примере используется понятие пространства имен **namespace**. Пространство имен введено для разделения наборов элементов с соответствующими правилами, описанными схемой. Пространство имен объявляется с помощью атрибута **xmlns** и префикса, который используется для элементов из данного пространства.

Например, **xmlns="http://www.w3.org/2001/XMLSchema"** задает пространство имен по умолчанию для элементов, атрибутов и типов схемы, которые принадлежат пространству имен **"http://www.w3.org/2001/XMLSchema"** и описаны соответствующей схемой.

Атрибут **targetNamespace="http://www.example.com/Students"** задает пространство имен для элементов/атрибутов, которые описывает данная схема.

Атрибут **xmlns:tns="http://www.example.com/Students"** вводит префикс для пространства имен (элементов) данной схемы. То есть для всех элементов, типов, описанных данной схемой и используемых здесь же требуется использовать префикс **tns**, как в случае с типами – **tns:Address**, **tns:Login** и т.д.

Действие пространства имён распространяется на элемент, где он объявлен, и на все дочерние элементы.

Тогда для проверки документа объекту-парсеру следует дать указание использовать DTD или схему XSD, и в XML-документ вместо ссылки на DTD добавить вместо корневого элемента **<students>** элемент **<tns:students>** вида:

```
<tns:students xmlns:tns="http://www.example.com/Students"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="http://www.example.com/Students
students.xsd ">
```

Следующий пример выполняет проверку документа на корректность средствами языка Java.

```
/* пример # 13 : проверка корректности документа XML: XSDMain.java */
package chapt16.xsd;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXNotRecognizedException;
import org.xml.sax.SAXNotSupportedException;
import chapt16.xsd.MyErrorHandler;

public class XSDMain {
      public static void main(String[] args) {

            String filename = "students.xml";
            DOMParser parser = new DOMParser();
            try {
            // установка обработчика ошибок
      parser.setErrorHandler(new MyErrorHandler("log.txt"));

            // установка способов проверки с использованием XSD
                  parser.setFeature(
"http://xml.org/sax/features/validation", true);
                  parser.setFeature(
"http://apache.org/xml/features/validation/schema", true);

                  parser.parse(filename);
            } catch (SAXNotRecognizedException e) {
                  e.printStackTrace();
            System.out.print("идентификатор не распознан");
            } catch (SAXNotSupportedException e) {
                  e.printStackTrace();
            System.out.print("неподдерживаемая операция");

            } catch (SAXException e) {
                  e.printStackTrace();
            System.out.print("глобальная SAX ошибка ");
            } catch (IOException e) {
                  e.printStackTrace();
            System.out.print("ошибка I/O потока");

            }
      System.out.print("проверка " + filename + " завершена");
      }
}
```

Класс обработчика ошибок может выглядеть следующим образом:

```
/* пример # 14 : обработчик ошибок : MyErrorHandler.java */
package chapt16.xsd;
import java.io.IOException;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class MyErrorHandler implements ErrorHandler {
   private Logger logger;

   public MyErrorHandler(String log) throws IOException {
      //создание регистратора ошибок chapt16.xsd
      logger = Logger.getLogger("chapt16.xsd");
      //установка файла и формата вывода ошибок
      logger.addAppender(new FileAppender(
                        new SimpleLayout(), log));
   }
   public void warning(SAXParseException e) {
      logger.warn(getLineAddress(e) + "-" +
         e.getMessage());
   }
   public void error(SAXParseException e) {
      logger.error(getLineAddress(e) + " - "
         + e.getMessage());
   }
   public void fatalError(SAXParseException e) {
      logger.fatal(getLineAddress(e) + " - "
         + e.getMessage());
   }
   private String getLineAddress(SAXParseException e) {
      //определение строки и столбца ошибки
      return e.getLineNumber() + " : "
         + e.getColumnNumber();
   }
}
```

Чтобы убедиться в работоспособности кода, следует внести в исходный XML-документ ошибку. Например, сделать идентичными значения атрибута **login**. Тогда в результате запуска в файл будут выведены следующие сообщения обработчика об ошибках:

```
ERROR - 14 : 41 - cvc-id.2: There are multiple occurrences
of ID value 'mit'.
ERROR - 14 : 41 - cvc-attribute.3: The value 'mit' of
attribute 'login' on element 'student' is not valid with
respect to its type, 'login'.
```

Если допустить синтаксическую ошибку в XML-документе, например, удалить закрывающую скобку в элементе **telephone**, будет выведено сообщение о фатальной ошибке:

FATAL - 7 : 26 - Element type "telephone2456474" must be followed by either attribute specifications, ">" or "/>".

В Java разработаны серьезные способы взаимодействия с XML. Начиная с версии Java 6, эти механизмы включены в JDK.

Следующий пример на основе внутреннего класса создает структуру документа XML и сохраняет в ней объект.

```
/* пример # 15 : создание XML-документа на основе объекта: DemoJSR.java */
package chapt16;
import java.io.*;
import javax.xml.bind.*;
import javax.xml.bind.annotation.*;

public class DemoJSR {
   public static void main(String[] args) {
      try {
         JAXBContext context =
                  JAXBContext.newInstance(Student.class);
         Marshaller m = context.createMarshaller();
         Student s = new Student(1, "Bender");//объект
            m.marshal(s, new FileOutputStream("stud.xml"));
      } catch (FileNotFoundException e) {
         System.out.println("XMl-файл не найден");
            e.printStackTrace();
      } catch (JAXBException e) {
         System.out.println("JAXB-исключения");
         e.printStackTrace();
      }
   }
   @XmlRootElement
   private static class Student {//внутренний класс
      private int id;
      private String name;

      public Student() {
      }
      public Student(int id, String name) {
         this.id = id;
         this.name = name;
      }
      public int getID() {
         return id;
      }
      public String getName() {
         return name;
      }
      public void setID(int id) {
         this.id = id;
      }
```

```
        public void setName(String name) {
            this.name = name;
        }
    }
}
```

В результате компиляции и запуска программы будет создан XML-документ :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<student>
      <ID>1</ID>
      <name>Bender</name>
</student>
```

Возможно обратное создание на основе XML-схемы классов на языке Java:

/ пример # 16 : описание классов University, Course и перечисления Faculty в XSD-схеме: student.xsd*/*

```
<schema xmlns="http://www.w3c.org/2001/XMLSchema"
xmlns:Revealed="http://www.university.net"
targetNamespace="http://www.university.net">

   <element name="University">
      <complexType>
         <sequence>
          <element name="faculty" type="Revealed:Faculty"/>
          <element name="course" type="Revealed:Course"/>
         </sequence>
      </complexType>
   </element>
   <complexType name="Course">
      <sequence>
         <element name="login" type="string"/>
         <element name="name" type="string"/>
         <element name="telephone" type="string"/>
      </sequence>
   </complexType>
   <simpleType name="Faculty">
      <restriction base="string">
         <enumeration value="FPMI"></enumeration>
         <enumeration value="MMF"></enumeration>
         <enumeration value="Geo"></enumeration>
      </restriction>
   </simpleType>
</schema>
```

Запуск выполняется с помощью командной строки:

```
xjc student.xsd
```

В результате будет сгенерирован следующий код классов:

```
package net.university;
import javax.xml.bind.annotation.XmlEnum;
import javax.xml.bind.annotation.XmlEnumValue;
@XmlEnum
```

```
public enum Faculty {
    FPMI("FPMI"),
    MMF("MMF"),
    @XmlEnumValue("Geo")
    GEO_F("Geo");
    private final String value;

    Faculty(String v) {
        value = v;
    }
    public String value() {
        return value;
    }
    public static Faculty fromValue(String v) {
        for (Faculty c: Faculty.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v.toString());
    }
}
package net.university;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
/**
 * <p>Java class for Course complex type.
*/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Course", propOrder = {
    "login",
    "name",
    "telephone"
})
public class Course {

    @XmlElement(required = true)
    protected String login;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String telephone;
    public String getLogin() {
        return login;
    }
    public void setLogin(String value) {
        this.login = value;
    }
```

```
    public String getName() {
        return name;
    }
    public void setName(String value) {
        this.name = value;
    }
    public String getTelephone() {
        return telephone;
    }
    public void setTelephone(String value) {
        this.telephone = value;
    }
}
package net.university;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
/**
 * <p>Java class for anonymous complex type.
*/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "faculty",
    "course"
})
@XmlRootElement(name = "University")
public class University {

    @XmlElement(required = true)
    protected Faculty faculty;
    @XmlElement(required = true)
    protected Course course;
    public Faculty getFaculty() {
        return faculty;
    }
    public void setFaculty(Faculty value) {
        this.faculty = value;
    }
    public Course getCourse() {
        return course;
    }
    public void setCourse(Course value) {
        this.course = value;
    }
}
package net.university;
import javax.xml.bind.annotation.XmlRegistry;
```

```
@XmlRegistry
public class ObjectFactory {
    public ObjectFactory() {
    }
    public Course createCourse() {
        return new Course();
    }
    public University createUniversity() {
        return new University();
    }
}
```

XML-анализаторы

XML как набор байт в памяти, запись в базе или текстовый файл представляет собой данные, которые еще предстоит обработать. То есть из набора строк необходимо получить данные, пригодные для использования в программе. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки – XML-анализаторы (парсеры).

Парсер – это библиотека (в языке Java: класс), которая читает XML-документ, а затем предоставляет набор методов для обработки информации этого документа.

Валидирующие и невалидирующие анализаторы

Как было выше упомянуто, существует два вида корректности XML-документа: синтаксическая (well-formed) – документ сформирован в соответствии с синтаксическими правилами построения, и действительная (valid) – документ синтаксически корректен и соответствует требованиям, заявленным в DTD.

Соответственно есть невалидирующие и валидирующие анализаторы. И те, и другие проверяют XML-документ на соответствие синтаксическим правилам. Но только валидирующие анализаторы знают, как проверить XML-документ на соответствие структуре, описанной в XSD или DTD.

Никакой связи между видом анализатора и видом XML-документа нет. Валидирующий анализатор может разобрать XML-документ, для которого нет DTD, и, наоборот, невалидирующий анализатор может разобрать XML-документ, для которого есть DTD. При этом он просто не будет учитывать описание структуры документа.

Древовидная и событийная модели

Существует три подхода (API) к обработке XML-документов:

- DOM (Document Object Model – объектная модель документов) – платформенно-независимый программный интерфейс, позволяющий программам и скриптам управлять содержимым документов HTML и XML, а также изменять их структуру и оформление. Модель DOM не накладывает ограничений на структуру документа. Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел которого содержит элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями родитель-потомок.
- SAX (Simple API for XML) базируется на модели последовательной одноразовой обработки и не создает внутренних деревьев. При прохожде-

нии по XML вызывает соответствующие методы у классов, реализующих интерфейсы, предоставляемые SAX-парсером.

- StAX (Streaming API for XML) не создает дерево объектов в памяти, но, в отличие от SAX-парсера, за переход от одной вершины XML к другой отвечает приложение, которое запускает разбор документа.

Анализаторы, которые строят древовидную модель, – это DOM-анализаторы. Анализаторы, которые генерируют события, – это SAX-анализаторы.

Анализаторы, которые ждут команды от приложения для перехода к следующему элементу XML – StAX-анализаторы.

В первом случае анализатор строит в памяти дерево объектов, соответствующее XML-документу. Далее вся работа ведется именно с этим деревом.

Во втором случае анализатор работает следующим образом: когда происходит анализ документа, анализатор вызывает методы, связанные с различными участками XML-файла, а программа, использующая анализатор, решает, как реагировать на тот или иной элемент XML-документа. Так, аннализатор будет генерировать событие о том, что он встретил начало документа либо его конец, начало элемента либо его конец, символьную информацию внутри элемента и т.д.

StAX работает как **`Iterator`**, который указывает на наличие элемента с помощью метода **`hasNext()`** и для перехода к следующей вершине использует метод **`next()`**.

Когда следует использовать DOM-, а когда – SAX, StAX -анализаторы?

DOM-анализаторы следует использовать тогда, когда нужно знать структуру документа и может понадобиться изменять эту структуру либо использовать информацию из XML-файла несколько раз.

SAX/StAX-анализаторы используются тогда, когда нужно извлечь информацию о нескольких элементах из XML-файла либо когда информация из документа нужна только один раз.

Событийная модель

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует события, когда находит элементы, атрибуты или текст. На первый взгляд, такой подход менее естествен для приложения, использующего анализатор, так как он не строит дерево, а приложение само должно догадаться, какое дерево элементов описывается в XML-файле.

Однако нужно учитывать, для каких целей используются данные из XML-файла. Очевидно, что нет смысла строить дерево объектов, содержащее десятки тысячи элементов в памяти, если всё, что необходимо, – это просто посчитать точное количество элементов в файле.

SAX-анализаторы

SAX API определяет ряд методов, используемых при разборе документа:

`void startDocument()` – вызывается на старте обработки документа;

`void endDocument()` – вызывается при завершении разбора документа;

`void startElement(String uri, String localName, String qName, Attributes attrs)` – будет вызван, когда анализатор полностью обработает содержимое открывающего тега, включая его имя и все содержащиеся атрибуты;

void endElement(String uri, String localName, String qName) – сигнализирует о завершении элемента;

void characters(char[] ch, int start, int length) – вызывается в том случае, если анализатор встретил символьную информацию внутри элемента (тело тега);

warning(SAXParseException e), **error(SAXParseException e)**, **fatalError(SAXParseException e)** – вызываются в ответ на возникающие предупреждения и ошибки при разборе XML-документа.

В пакете **org.xml.sax в SAX2 API** содержатся интерфейсы **org.xml.sax.ContentHandler**, **org.xml.sax.ErrorHandler**, **org.xml.sax.DTDHandler**, и **org.xml.sax.EntityResolver**, которые необходимо реализовать для обработки соответствующего события.

Для того чтобы создать простейшее приложение, обрабатывающее XML-документ, достаточно сделать следующее:

1. Создать класс, который реализует один или несколько интерфейсов (**ContentHandler**, **ErrorHandler**, **DTDHandler**, **EntityResolver**) и реализовать методы, отвечающие за обработку интересующих событий.
2. Используя SAX2 API, поддерживаемое всеми SAX парсерами, создать **org.xml.sax.XMLReader**, например для Xerces:
   ```
   XMLReader reader =
   XMLReaderFactory.createXMLReader(
       "org.apache.xerces.parsers.SAXParser");
   ```
3. Передать в **XMLReader** объект класса, созданного на шаге 1 с помощью соответствующих методов:
   setContentHandler(), **setErrorHandler()**, **setDTDHandler()**, **setEntityResolver()** .
4. Вызвать метод **parse()**, которому в качестве параметров передать путь (URI) к анализируемому документу либо **InputSource**.

Следующий пример выводит на консоль содержимое XML-документа.

```
/* пример # 1 : чтение и вывод XML-документа : SimpleHandler.java */
package chapt16.analyzer.sax;
import org.xml.sax.ContentHandler;
import org.xml.sax.Attributes;

public class SimpleHandler implements ContentHandler {

   public void startElement(String uri, String localName,
         String qName, Attributes attrs) {
      String s = qName;
      //получение и вывод информации об атрибутах элемента
      for (int i = 0; i < attrs.getLength(); i++) {
         s += " " + attrs.getQName(i) + "="
            + attrs.getValue(i) + " ";
      }
      System.out.print(s.trim());
   }
```

```
   public void characters(char[] ch,
         int start, int length) {
      System.out.print(new String(ch, start, length));
   }
   public void endElement(String uri,
      String localName, String qName) {
      System.out.print(qName);
   }
}
```

/ пример # 2 : создание и запуск парсера : SAXSimple.java*/*

```
package chapt16.main;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLReaderFactory;
import org.xml.sax.SAXException;
import javax.xml.parsers.ParserConfigurationException;
import java.io.IOException;
import chapt16.analyzer.sax.SimpleHandler;

public class SAXSimple {
   public static void main(String[] args) {
      try {
         //создание SAX-анализатора
XMLReader reader = XMLReaderFactory.createXMLReader();
        SimpleHandler contentHandler = new SimpleHandler();
        reader.setContentHandler(contentHandler);
        reader.parse("students.xml");
      } catch (SAXException e) {
         e.printStackTrace();
         System.out.print("ошибка SAX парсера");
      } catch (ParserConfigurationException e) {
         e.printStackTrace();
         System.out.print("ошибка конфигурации");
      } catch (IOException e) {
         e.printStackTrace();
         System.out.print("ошибка I/O потока");
      }
   }
}
```

В результате в консоль будет выведено (если убрать из XML-документа ссылку на DTD):

```
students
      student login=mit faculty=mmf
            name Mitar Alex name
            telephone 2456474 telephone
            address
                  country Belarus country
                  city Minsk city
                  street Kalinovsky 45 street
             address
```

```
       student
      student login=pus faculty=mmf
            name Pashkun Alex name
            telephone 3453789 telephone
            address
                  country Belarus country
                  city Brest city
                  street Knorina 56 street
             address
       student
students
```

В следующем приложении производятся разбор документа **students.xml** и инициализация на его основе коллекции объектов класса **Student**.

/ пример # 3 : формирование коллекции объектов на основе XML-документа : StudentHandler.java */*

```
package chapt16.analyzer.sax;

enum StudentEnum {
   NAME, TELEPHONE, STREET, CITY, COUNTRY
}
package chapt16.analyzer.sax;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import java.util.ArrayList;
import chapt16.entity.Student;

public class StundentHandler implements ContentHandler {
   ArrayList<Student> students = new ArrayList<Student>();
   Student curr = null;
   StudentEnum currentEnum = null;

   public ArrayList<Student> getStudents() {
      return students;
   }
   public void startDocument() {
      System.out.println("parsing started");
   }
   public void startElement(String uri, String localName,
         String qName, Attributes attrs) {
      if (qName.equals("student")) {
         curr = new Student();
         curr.setLogin(attrs.getValue(0));
         curr.setFaculty(attrs.getValue(1));
      }
      if(!"address".equals(qName) &&
         !"student".equals(qName) &&
         !qName.equals("students"))
         currentEnum =
            StudentEnum.valueOf(qName.toUpperCase());
   }
```

```
   public void endElement(String uri, String localName,
         String qName) {
      if(qName.equals("student"))
         students.add(curr);
      currentEnum = null;
   }
   public void characters(char[] ch, int start,
                                          int length) {
      String s = new String(ch, start, length).trim();
      if(currentEnum == null) return;
      switch (currentEnum) {
         case NAME:
            curr.setName(s);
            break;
         case TELEPHONE:
            curr.setTelephone(s);
            break;
         case STREET:
            curr.getAddress().setStreet(s);
            break;
         case CITY:
            curr.getAddress().setCity(s);
            break;
         case COUNTRY:
            curr.getAddress().setCountry(s);
            break;
      }
   }
}
```

/ пример # 4 : создание и запуск парсера : SAXStudentMain.java */*

```
package chapt16.main;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLReaderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import java.util.ArrayList;
import chapt16.analyzer.sax.StundentHandler;
import chapt16.entity.Student;
import java.io.IOException;

public class SAXStudentMain {
   public static void main(String[] args) {
      try {
         //создание SAX-анализатора
        XMLReader reader =
            XMLReaderFactory.createXMLReader();
         StundentHandler sh = new StundentHandler();
        reader.setContentHandler(sh);
```

```
            ArrayList <Student> list;
            if(sh != null) {
               //разбор XML-документа
               parser.parse("students.xml");
                  System.out.println(sh.getStudents());
               }
         } catch (SAXException e) {
            e.printStackTrace();
            System.out.print("ошибка SAX парсера");
         } catch (ParserConfigurationException e) {
            e.printStackTrace();
            System.out.print("ошибка конфигурации");
         } catch (IOException e) {
            e.printStackTrace();
            System.out.print("ошибка I/O потока");
         }
      }
}
```

В результате на консоль будет выведена следующая информация:

```
parsing started
Login: mit
Name: Mitar Alex
Telephone: 2456474
Faculty: mmf
Address:
        Country: Belarus
        City: Minsk
        Street: Kalinovsky 45
Login: pus
Name: Pashkun Alex
Telephone: 3453789
Faculty: mmf
Address:
        Country: Belarus
        City: Brest
        Street: Knorina 56
```

Класс, объект которого формируется на основе информации из XML-документа, имеет следующий вид:

```
/* пример # 5 : класс java bean : Student.java */
package chapt16.entity;

public class Student {
   private String login;
   private String name;
   private String faculty;
   private String telephone;
   private Address address = new Address();
```

```
public String getLogin() {
   return login;
}
public void setLogin(String login) {
   this.login = login;
}
public String getName() {
   return name;
}
public void setName(String name) {
   this.name = name;
}
public String getFaculty() {
   return faculty;
}
public void setFaculty(String faculty) {
   this.faculty = faculty;
}
public String getTelephone() {
   return telephone;
}
public void setTelephone(String telephone) {
   this.telephone = telephone;
}
public Address getAddress() {
   return address;
}
public void setAddress(Address address) {
   this.address = address;
}
public String toString() {
   return "Login: " + login
      + "\nName: " + name
      + "\nTelephone: " + telephone
      + "\nFaculty: " + faculty
      + "\nAddress:"
      + "\n\tCountry: " + address.getCountry()
      + "\n\tCity: " + address.getCity()
      + "\n\tStreet: " + address.getStreet()
      + "\n";
}
public class Address {//внутренний класс
   private String country;
   private String city;
   private String street;

   public String getCountry() {
      return country;
   }
```

```
        public void setCountry(String country) {
            this.country = country;
        }
        public String getCity() {
            return city;
        }
        public void setCity(String city) {
            this.city = city;
        }
        public String getStreet() {
            return street;
        }
        public void setStreet(String street) {
            this.street = street;
        }
    }
}
```

Древовидная модель

Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода.

DOM строит дерево, которое представляет содержимое XML-документа, и определяет набор классов, которые представляют каждый элемент в XML-документе (элементы, атрибуты, сущности, текст и т.д.).

В пакете **org.w3c.dom** можно найти интерфейсы, которые представляют вышеуказанные объекты. Реализацией этих интерфейсов занимаются разработчики анализаторов. Разработчики приложений, которые хотят использовать DOM-анализатор, имеют готовый набор методов для манипуляции деревом объектов и не зависят от конкретной реализации используемого анализатора.

Существуют различные общепризнанные DOM-анализаторы, которые в настоящий момент можно загрузить с указанных адресов:

Xerces – http://xerces.apache.org/xerces2-j/;

JAXP – входит в JDK.

Существуют также библиотеки, предлагающие свои структуры объектов XML с API для доступа к ним. Наиболее известные:

JDOM – http://www.jdom.org/dist/binary/jdom-1.0.zip.

dom4j – http://www.dom4j.org

Xerces

В стандартную конфигурацию Java входит набор пакетов для работы с XML. Но стандартная библиотека не всегда является самой простой в применении, поэтому часто в основе многих проектов, использующих XML, лежат библиотеки сторонних производителей. Одной из таких библиотек является Xerces, замечательной особенностью которого является использование части стандартных возможностей XML-библиотек JSDK с добавлением собственных классов и методов, упрощающих и облегчающих обработку документов XML.

org.w3c.dom.Document

Используется для получения информации о документе и изменения его структуры. Это интерфейс представляет собой корневой элемент XML-документа и содержит методы доступа ко всему содержимому документа.

`Element getDocumentElement()` – возвращает корневой элемент документа.

org.w3c.dom.Node

Основным объектом DOM является **`Node`** – некоторый общий элемент дерева. Большинство DOM-объектов унаследовано именно от **`Node`**. Для представления элементов, атрибутов, сущностей разработаны свои специализации **`Node`**.

Интерфейс **`Node`** определяет ряд методов, которые используются для работы с деревом:

`short getNodeType()` – возвращает тип объекта (элемент, атрибут, текст, **`CDATA`** и т.д.);

`String getNodeValue()` – возвращает значение **`Node`**;

`Node getParentNode()` – возвращает объект, являющийся родителем текущего узла **`Node`**;

`NodeList getChildNodes()` – возвращает список объектов, являющихся дочерними элементами;

`Node getFirstChild()`, **`Node getLastChild()`** – возвращает первый и последний дочерние элементы;

`NamedNodeMap getAttributes()` – возвращает список атрибутов данного элемента.

У интерфейса **`Node`** есть несколько важных наследников – **`Element`**, **`Attr`**, **`Text`**. Они используются для работы с конкретными объектами дерева.

org.w3c.dom.Element

Интерфейс предназначен для работы с содержимым элементов XML-документа. Некоторые методы:

`String getTagName(String name)` – возвращает имя элемента;

`boolean hasAttribute()` – проверяет наличие атрибутов;

`String getAttribute(String name)` – возвращает значение атрибута по его имени;

`Attr getAttributeNode(String name)` – возвращает атрибут по его имени;

`void setAttribute(String name, String value)` – устанавливает значение атрибута, если необходимо, атрибут создается;

`void removeAttribute(String name)` – удаляет атрибут;

`NodeList getElementsByTagName(String name)` – возвращает список дочерних элементов с определенным именем.

org.w3c.dom.Attr

Интерфейс служит для работы с атрибутами элемента XML-документа.

Некоторые методы интерфейса **`Attr`**:

`String getName()` – возвращает имя атрибута;

`Element getOwnerElement` – возвращает элемент, который содержит этот атрибут;

String getValue() – возвращает значение атрибута;
void setValue(String value) – устанавливает значение атрибута;
boolean isId() – проверяет атрибут на тип ID.

org.w3c.dom.Text

Интерфейс **Text** необходим для работы с текстом, содержащимся в элементе.
String getWholeText() – возвращает текст, содержащийся в элементе;
void replaceWholeText(String content) – заменяет строкой **content** весь текст элемента.

В следующих примерах производятся разбор документа **students.xml** с использованием DOM-анализатора и инциализация на его основе набора объектов.

```
/* пример # 6 : создание анализатора и загрузка XML-документа:
DOMLogic.java*/
package chapt16.main;
import java.util.ArrayList;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
//import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.SAXException;
import chapt16.analyzer.dom.Analyzer;
import chapt16.entity.Student;

public class DOMLogic {
      public static void main(String[] args) {
            try {
            // создание DOM-анализатора(JSDK)
            DocumentBuilderFactory dbf=
                  DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            // распознавание XML-документа
            Document document = db.parse("students.xml");

             // создание DOM-анализатора (Xerces)
            /* DOMParser parser = new DOMParser();
            parser.parse("students.xml");
            Document document = parser.getDocument();*/

            Element root = document.getDocumentElement();
ArrayList<Student> students = Analyzer.listBuilder(root);

            for (int i = 0; i < students.size(); i++) {
                  System.out.println(students.get(i));
                  }
```

```
      } catch (SAXException e) {
         e.printStackTrace();
         System.out.print("ошибка SAX парсера");
      } catch (ParserConfigurationException e) {
         e.printStackTrace();
         System.out.print("ошибка конфигурации");
      } catch (IOException e) {
         e.printStackTrace();
         System.out.print("ошибка I/O потока");
            }
      }
}
```

/ пример # 7 : создание объектов на основе объекта типа Element : Analyzer.java */*

```
package chapt16.analyzer.dom;
import java.util.ArrayList;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import chapt16.entity.Student;

public class Analyzer {
 public static ArrayList<Student> listBuilder(Element root)
                  throws SAXException, IOException {
            ArrayList<Student> students
                              = new ArrayList<Student>();
                  // получение списка дочерних элементов <student>
                  NodeList studentsNodes =
                      root.getElementsByTagName("student");
                  Student student = null;
      for (int i = 0; i < studentsNodes.getLength(); i++) {
            student = new Student();
            Element studentElement =
                        (Element) studentsNodes.item(i);
                        // заполнение объекта student
student.setFaculty(studentElement.getAttribute("faculty"));
student.setName(getBabyValue(studentElement, "name"));
student.setLogin(studentElement.getAttribute("login"));
student.setTelephone(
            getBabyValue(studentElement, "telephone"));

            Student.Address address = student.getAddress();
                        // заполнение объекта address
            Element addressElement =
                  getBaby(studentElement, "address");
            address.setCountry(
                getBabyValue(addressElement, "country"));
```

```
            address.setCity(
                  getBabyValue(addressElement, "city"));
            address.setStreet(
                  getBabyValue(addressElement, "street"));

                  students.add(student);
                  }
            return students;
      }
      // возвращает дочерний элемент по его имени и родительскому элементу
      private static Element getBaby(Element parent,
                                    String childName) {
            NodeList nlist =
                  parent.getElementsByTagName(childName);
            Element child = (Element) nlist.item(0);
            return child;
      }
      // возвращает текст, содержащийся в элементе
      private static String getBabyValue(Element parent,
                                    String childName) {
            Element child = getBaby(parent, childName);
            Node node = child.getFirstChild();
            String value = node.getNodeValue();
            return value;
      }
}
```

JDOM

JDOM не является анализатором, он был разработан для более удобного, более интуитивного для Java-программист, доступа к объектной модели XML-документа. JDOM представляет свою модель, отличную от DOM. Для разбора документа JDOM использует либо SAX-, либо DOM-парсеры сторонних производителей. Реализаций JDOM немного, так как он основан на классах, а не на интерфейсах.

Разбирать XML-документы с помощью JDOM проще, чем с помощью Xerces. Иерархия наследования объектов документа похожа на Xerces.

Content

В корне иерархии наследования стоит класс **`Content`**, от которого унаследованы остальные классы (**`Text`**, **`Element`** и др.).

Основные методы класса **`Content`**:

`Document getDocument()` – возвращает объект, в котором содержится этот элемент;

`Element getParentElement()` – возвращает родительский элемент.

Document

Базовый объект, в который загружается после разбора XML-документ. Аналогичен **`Document`** из Xerces.

`Element getRootElement()` – возвращает корневой элемент.

Parent

Интерфейс **Parent** реализуют классы **Document** и **Element**. Он содержит методы для работы с дочерними элементами. Интерфейс **Parent** и класс **Content** реализуют ту же функциональность, что и интерфейс **Node** в Xerces.

Некоторые из его методов:

List getContent() – возвращает все дочерние объекты;

Content getContent(int index) – возвращает дочерний элемент по его индексу;

int getContentSize() – возвращает количество дочерних элементов;

Parent getParent() – возвращает родителя этого родителя;

int indexOf(Content child) – возвращает индекс дочернего элемента.

Element

Класс **Element** представляет собой элемент XML-документа.

Attribute getAttribute(String name) – возвращает атрибут по его имени;

String getAttributeValue(String name) – возвращает значение атрибута по его имени;

List getAttributes() – возвращает список всех атрибутов;

Element getChild(String name) – возвращает дочерний элемент по имени;

List getChildren() – возвращает список всех дочерних элементов;

String getChildText(String name) – возвращает текст дочернего элемента;

String getName() – возвращает имя элемента;

String getText() – возвращает текст, содержащийся в элементе.

Text

Класс **Text** содержит методы для работы с текстом. Аналог в Xerces – интерфейс **Text**.

String getText() – возвращает значение содержимого в виде строки;

String getTextTrim() – возвращает значение содержимого без крайних пробельных символов.

Attribute

Класс **Attribute** представляет собой атрибут элемента XML-документа. В отличие от интерфейса **Attr** из Xerces, у класса **Attribute** расширенная функциональность. Класс **Attribute** имеет методы для возвращения значения определенного типа.

int getAttributeType() – возвращает тип атрибута;

тип **get**Тип**Type()** – (**Int**, **Double**, **Boolean**, **Float**, **Long**) возвращает значение определенного типа;

String getName() – возвращает имя атрибута;

Element getParent() – возвращает родительский элемент.

Следующие примеры выполняют ту же функцию, что и предыдущие, только с помощью JDOM.

```
/* пример # 8 : запуск JDOM : JDOMStudentMain.java */
package chapt16.main;
import java.util.List;
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.IOException;
import chapt16.analyzer.dom.JDOMAnalyzer;
import chapt16.entity.Student;

public class JDOMStudentMain {
   public static void main(String[] args) {
      try {
      //создание JDOM
      SAXBuilder builder = new SAXBuilder();
      //распознавание XML-документа
      Document document = builder.build("students.xml");
         List<Student> list =
            JDOMAnalyzer.listCreator(document);

      for (Student st : list) System.out.println(st);
      } catch(IOException e) {
         e.printStackTrace();
      } catch(JDOMException e) {
         e.printStackTrace();
      }
   }
}
/* пример # 9 : создание объектов с использованием JDOM: JDOMAnalyzer.java */
package chapt16.analyzer.dom;
import java.util.*;
import java.io.IOException;
import org.jdom.Element;
import org.jdom.Document;
import org.jdom.JDOMException;
import chapt16.entity.Student;

public class JDOMAnalyzer {
   public static List<Student> listCreator(Document doc)
         throws JDOMException, IOException {
      //извлечение корневого элемента
      Element root = doc.getRootElement();
      //получение списка дочерних элементов <student>
      List studElem = root.getChildren();
      Iterator studentIterator = studElem.iterator();
      //создание пустого списка объектов типа Student
      ArrayList<Student> students =
                  new ArrayList<Student>();
      while(studentIterator.hasNext()) {
```

```
            Element studentElement =
               (Element)studentIterator.next();
            Student student = new Student();
            //заполнение объекта student
            student.setLogin(
         studentElement.getAttributeValue("login"));
            student.setName(
         studentElement.getChild("name").getText());
            student.setTelephone(
         studentElement.getChild("telephone").getText());
            student.setFaculty(
         studentElement.getAttributeValue("faculty"));

            Element addressElement =
               studentElement.getChild("address");
            Student.Address address = student.getAddress();
            //заполнение объекта address
address.setCountry(addressElement.getChild("country")
                                             .getText());
address.setCity(addressElement.getChild("city").getText());
address.setStreet(addressElement.getChild("street")
                                             .getText());
            students.add(student);
         }
         return students;
      }
}
```

Создание и запись XML-документов

Документы можно не только читать, но также модифицировать и создавать совершенно новые.

Для создания документа необходимо создать объект каждого класса (**Element**, **Attribute**, **Document**, **Text** и др.) и присоединить его к объекту, который в дереве XML-документа находится выше. В данном разделе будет рассматриваться только анализатор JDOM.

Element

Для добавления дочерних элементов, текста или атрибутов в элемент XML-документа нужно использовать один из следующих методов:

Element addContent(Content child) – добавляет дочерний элемент;

Element addContent(int index, Content child) – добавляет дочерний элемент в определенную позицию;

Element addContent(String str) – добавляет текст в содержимое элемента;

Element setAttribute(Attribute attribute) – устанавливает значение атрибута;

Element setAttribute(String name, String value) – также устанавливает значение атрибута;

Element setContent(Content child) – заменяет содержимое этого элемента на элемент, переданный в качестве параметра;

Element setContent(int index, Content child) – заменяет дочерний элемент на определенной позиции элементом, переданным как параметр;

Element setName(String name) – устанавливает имя элемента;

Element setText(String text) – устанавливает текст содержимого элемента.

Text

Класс **Text** также имеет методы для добавления текста в элемент XML-документа:

void append(String str) – добавляет текст к уже имеющемуся;

void append(Text text) – добавляет текст из другого объекта **Text**, переданного в качестве параметра;

Text setText(String str) – устанавливает текст содержимого элемента.

Attribute

Методы класса **Attribute** для установки значения, имени и типа атрибута:

Attribute setAttributeType(int type) – устанавливает тип атрибута;

Attribute setName(String name) – устанавливает имя атрибута;

Attribute setValue(String value) – устанавливает значение атрибута.

Следующий пример демонстрирует создание XML-документа и запись его в файл. Для записи XML-документа используется класс **XMLOutputter**.

```
/* пример # 10 : создание и запись документа с помощью JDOM:
JDOMLogic.java */
package chapt16.saver.dom;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.List;
import java.util.Iterator;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.XMLOutputter;
import chapt16.entity.Student;

public class JDOMLogic {
      public static Document create(List<Student> list) {
//создание корневого элемента <studentsnew>
      Element root = new Element("studentsnew");
      Iterator<Student> studentIterator =
         list.iterator();
      while(studentIterator.hasNext()) {
         Student student = studentIterator.next();
         //создание элемента <student> и его содержимого
         Element studentElement = new Element("student");
         //создание атрибутов  и передача им значений
         studentElement.setAttribute("login",
            student.getLogin());
```

```
            studentElement.setAttribute("phone",
               student.getTelephone());

               Element faculty = new Element("faculty");
               faculty.setText(student.getFaculty());
            //«вложение» элемента <faculty> в элемент <student>
               studentElement.addContent(faculty);

               Element name = new Element("name");
               name.setText(student.getName());
               studentElement.addContent(name);
            //создание элемента <address>
            Element addressElement = new Element("address");
            Student.Address address = student.getAddress();

            Element country = new Element("country");
            country.setText(address.getCountry());
            addressElement.addContent(country);

            Element city = new Element("city");
            city.setText(address.getCity());
            addressElement.addContent(city);

            Element street = new Element("street");
            street.setText(address.getStreet());
            // «вложение» элемента <street> в элемент <address>
            addressElement.addContent(street);
            //«вложение» элемента <address> в элемент <student>
            studentElement.addContent(addressElement);
            //«вложение» элемента <student> в элемент <students>
            root.addContent(studentElement);
         }
            //создание основного дерева XML-документа
         return new Document(root);
      }
public static boolean saveDocument(String fileName,
                                   Document doc) {
            boolean complete = true;
            XMLOutputter outputter = new XMLOutputter();
            // запись XML-документа
            try {
outputter.output(doc, new FileOutputStream(fileName));
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
                  complete = false;
            } catch (IOException e) {
                  e.printStackTrace();
                  complete = false;
            }
```

```
            return complete;
      }
}
```

/ пример # 11 : создание списка и запуск приложения : JDOMMainSaver.java*/*

```
package chapt16.main;
import java.io.IOException;
import java.util.ArrayList;
import chapt16.entity.Student;
import chapt16.saver.dom.JDOMLogic;

public class JDOMMainSaver {
   public static void main(String[] args) {
         //создание списка студентов
ArrayList<Student> students = new ArrayList<Student> ();
         for(int j = 1; j < 3; j++) {
            Student st = new Student();
            st.setName("Petrov" + j);
            st.setLogin("petr" + j);
            st.setFaculty("mmf");
            st.setTelephone("454556"+ j*3);
          Student.Address adr = st.getAddress();
            adr.setCity("Minsk");
            adr.setCountry("BLR");
            adr.setStreet("Gaja, " + j);
                  st.setAddress(adr);
                        students.add(st);
         }
         //создание «дерева» на основе списка студентов
         Document doc = JDOMLogic.create(students);
         //сохранение «дерева» в XML-документе
      if(JDOMLogic.saveDocument("studentsnew.xml", doc))
            System.out.println("Документ создан");
      else
            System.out.println("Документ НЕ создан");
   }
}
```

В результате будет создан документ **studentsnew.xml** следующего содержания:

```
<?xml version="1.0" encoding="UTF-8"?>
<studentsnew>
      <student login="petr1" phone="4545563">
            <faculty>mmf</faculty>
            <name>Petrov1</name>
            <address>
                  <country>BLR</country>
                  <city>Minsk</city>
                  <street>Gaja, 1</street>
            </address>
      </student>
```

```
    <student login="petr2" phone="4545566">
          <faculty>mmf</faculty>
          <name>Petrov2</name>
          <address>
                <country>BLR</country>
                <city>Minsk</city>
                <street>Gaja, 2</street>
          </address>
    </student>
</studentsnew>
```

В этом примере был использован JDOM, основанный на идее "if something doesn't work, fix it".

StAX

StAX (Streaming API for XML), который еще называют pull-парсером, включен в JDK, начиная с версии Java SE 6. Он похож на SAX отсутствием объектной модели в памяти и последовательным продвижением по XML, но в StAX не требуется реализация интерфейсов, и приложение само командует StAX-парсеру перейти к следующему элементу XML. Кроме того, в отличие от SAX, данный парсер предлагает API для создания XML-документа.

Основными классами StAX являются **XMLInputFactory**, **XMLStreamReader** и **XMLOutputFactory**, **XMLStreamWriter**, которые соответственно используются для чтения и создания XML-документа. Для чтения XML надо получить ссылку на **XMLStreamReader**:

```
StringReader stringReader = new StringReader(xmlString);
XMLInputFactory inputFactory=XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory
                  .createXMLStreamReader(stringReader);
```

после чего **XMLStreamReader** можно применять аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **next()**:

boolean hasNext() – показывает, есть ли еще элементы;

int next() – переходит к следующей вершине XML, возвращая ее тип.

Возможные типы вершин:

XMLStreamConstants.START_DOCUMENT
XMLStreamConstants.END_DOCUMENT
XMLStreamConstants.START_ELEMENT
XMLStreamConstants.END_ELEMENT
XMLStreamConstants.CHARACTERS
XMLStreamConstants.ATTRIBUTE
XMLStreamConstants.CDATA
XMLStreamConstants.NAMESPACE
XMLStreamConstants.COMMENT
XMLStreamConstants.ENTITY_DECLARATION

Далее данные извлекаются применением методов:

String getLocalName() – возвращает название тега;

String getAttributeValue(NAMESPACE_URI, ATTRIBUTE_NAME) – возвращает значение атрибута;

String getText() – возвращает текст тега.

Пусть дан XML-документ с описанием медиатехники.

```
<?xml version="1.0" encoding="UTF-8"?>
<products xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation=" products.xsd">
      <category name="Audio And Video">
            <subcategory name="Audio">
                  <product>
                        <producer>Samsung</producer>
                        <model>NV678</model>
                        <year>12-12-2006</year>
                        <color>White</color>
                        <notAvailable />
                  </product>
            </subcategory>
            <subcategory name="Video">
                  <product>
                        <producer>Samsung</producer>
                        <model>VH500</model>
                        <year>12-12-2004</year>
                        <color>Black</color>
                        <cost>200</cost>
                  </product>
                  <product>
                        <producer>Samsung</producer>
                        <model>VH500</model>
                        <year>12-12-2004</year>
                        <color>White</color>
                        <notAvailable />
                  </product>
            </subcategory>
      </category>
      <category name="Computers">
            <subcategory name="Pocket">
                  <product>
                        <producer>HP</producer>
                        <model>rx371</model>
                        <year>31-01-2006</year>
                        <color>Black</color>
                        <notAvailable />
                  </product>
            </subcategory>
      </category>
</products>
```

Организация процесса разбора документа XML с помощью StAX приведена в следующем примере:

```
/* пример # 12 : реализация разбора XM-документа : StAXProductParser.java :
ProductParser.java: ParserEnum.java */
package chapt16;
```

```
public enum ParserEnum {
      PRODUCTS, CATEGORY, SUBCATEGORY, PRODUCT, PRODUCER,
MODEL, YEAR, COLOR, NOTAVAILABLE, COST, NAME
}
package chapt16;
import java.io.InputStream;

public abstract class ProductParser {
      public abstract void parse(InputStream input);

      public void writeTitle() {
            System.out.println("Products:");
      }
      public void writeCategoryStart(String name) {
            System.out.println("Category: " + name.trim());
      }
      public void writeCategoryEnd() {
            System.out.println();
      }
      public void writeSubcategoryStart(String name) {
         System.out.println("Subcategory: " + name.trim());
      }
      public void writeSubcategoryEnd() {
            System.out.println();
      }
      public void writeProductStart() {
            System.out.println("  Product Start  ");
      }
      public void writeProductEnd() {
            System.out.println("   Product End   ");
      }
      public void writeProductFeatureStart(String name) {
         switch (ParserEnum.valueOf(name.toUpperCase())) {
            case PRODUCER:
                  System.out.print("Provider: ");
                  break;
            case MODEL:
                  System.out.print("Model: ");
                  break;
            case YEAR:
                  System.out.print("Date of issue: ");
                  break;
            case COLOR:
                  System.out.print("Color: ");
                  break;
            case NOTAVAILABLE:
                  System.out.print("Not available");
                  break;
```

```
            case COST:
                  System.out.print("Cost: ");
                  break;
            }
         }
      public void writeProductFeatureEnd() {
            System.out.println();
      }

      public void writeText(String text) {
            System.out.print(text.trim());
      }
}
package chapt16;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
import java.io.InputStream;

public class StAXProductParser extends ProductParser {
      // реализация абстрактного метода из суперкласса для разбора потока
      public void parse(InputStream input) {
            XMLInputFactory inputFactory =
                        XMLInputFactory.newInstance();
            try {
               XMLStreamReader reader =
                 inputFactory.createXMLStreamReader(input);
               process(reader);
            } catch (XMLStreamException e) {
               e.printStackTrace();
            }
      }
      // метод, управляющий разбором потока
      public void process(XMLStreamReader reader)
                              throws XMLStreamException {
        String name;

        while (reader.hasNext()) {
       // определение типа "прочтённого" элемента (тега)
          int type = reader.next();

          switch (type) {
            case XMLStreamConstants.START_ELEMENT:
              name = reader.getLocalName();

      switch (ParserEnum.valueOf(name.toUpperCase())) {
            case PRODUCTS:
```

```
                writeTitle();
                break;
          case CATEGORY:
    writeCategoryStart(reader.getAttributeValue(null,
          ParserEnum.NAME.name().toLowerCase()));
          break;
          case SUBCATEGORY:
    writeSubcategoryStart(reader.getAttributeValue(null,
          ParserEnum.NAME.name().toLowerCase()));
          break;
          case PRODUCT:
                writeProductStart();
                break;
          default:
                writeProductFeatureStart(name);
                break;
    }
    break;

          case XMLStreamConstants.END_ELEMENT:
                name = reader.getLocalName();

    switch (ParserEnum.valueOf(name.toUpperCase())) {
          case CATEGORY:
                writeCategoryEnd();
                break;
          case SUBCATEGORY:
                writeSubcategoryEnd();
                break;
          case PRODUCT:
                writeProductEnd();
                break;
          default:
                writeProductFeatureEnd();
                break;
    }
    break;

          case XMLStreamConstants.CHARACTERS:
                writeText(reader.getText());
                break;

          default:
                break;
          }
      }
    }
}
```

Для запуска приложения разбора документа с помощью StAX ниже приведен достаточно простой код:

```
/* пример # 13 : запуск приложения : StreamOutputExample.java*/
package chapt16;
import java.io.FileInputStream;
import java.io.InputStream;

public class StreamOutputExample {
  public static void main(String[] args) throws Exception {
      ProductParser parser = new StAXProductParser();
      // создание входного потока данных из xml-файла
      InputStream input =
            new FileInputStream("chapt16\\mediatech.xml");
      // разбор файла с выводом результата на консоль
      parser.parse(input);
  }
}
```

XSL

Документ XML используется для представления информации в виде некоторой структуры, но он никоим образом не указывает, как его отображать. Для того чтобы просмотреть XML-документ, нужно его каким-то образом отформатировать. Инструкции форматирования XML-документов формируются в так называемые таблицы стилей, и для просмотра документа нужно обработать XML-файл согласно этим инструкциям.

Существует два стандарта стилевых таблиц, опубликованных W3C. Это CSS (Cascading Stylesheet) и XSL (XML Stylesheet Language).

CSS изначально разрабатывался для HTML и представляет из себя набор инструкций, которые указывают браузеру, какой шрифт, размер, цвет использовать для отображения элементов HTML-документа.

XSL более современен, чем CSS, потому что используется для преобразования XML-документа перед отображением. Так, используя XSL, можно построить оглавление для XML-документа, представляющего книгу.

Вообще XSL можно разделить на три части: XSLT (XSL Transformation), XPath и XSLFO (XSL Formatting Objects).

XSL Processor необходим для преобразования XML-документа согласно инструкциям, находящимся в файле таблицы стилей.

XSLT

Этот язык для описания преобразований XML-документа применяется не только для приведения XML-документов к некоторому “читаемому” виду, но и для изменения структуры XML-документа.

К примеру, XSLT можно использовать для:

- удаления существующих или добавления новых элементов в XML-документ;
- создания нового XML-документа на основании заданного;
- извлечения информации из XML-документа с разной степенью детализации;

- преобразования XML-документа в документ HTML или документ другого типа.

Пусть требуется построить новый XML-файл на основе файла **students.xml**, у которого будет удален атрибут **login**. Элементы **country**, **city**, **street** станут атрибутами элемента **address** и элемент **telephone** станет дочерним элементом элемента **address**. Следует воспользоваться XSLT для решения данной задачи. В следующем коде приведено содержимое файла таблицы стилей **students.xsl**, решающее поставленную задачу.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" />

   <xsl:template match="/">
      <students>
         <xsl:apply-templates />
      </students>
   </xsl:template>

   <xsl:template match="student">
      <xsl:element name="student">
         <xsl:attribute name="faculty">
            <xsl:value-of select="@faculty"/>
         </xsl:attribute>
         <name><xsl:value-of select="name"/></name>
         <xsl:element name="address">
            <xsl:attribute name="country">
               <xsl:value-of select="address/country"/>
            </xsl:attribute>
            <xsl:attribute name="city">
               <xsl:value-of select="address/city"/>
            </xsl:attribute>
            <xsl:attribute name="street">
               <xsl:value-of select="address/street"/>
            </xsl:attribute>
            <xsl:element name="telephone">
               <xsl:attribute name="number">
                  <xsl:value-of select="telephone"/>
               </xsl:attribute>
            </xsl:element>
         </xsl:element>
      </xsl:element>
   </xsl:template>
</xsl:stylesheet>
```

Преобразование XSL лучше сделать более коротким, используя ATV (attribute template value), т.е «**{}**»

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

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" />
   <xsl:template match="/">
      <students>
         <xsl:apply-templates />
</students>
   </xsl:template>
   <xsl:template match="student">
<student faculty="{@faculty}">
         <name><xsl:value-of select="name"/></name>
      <address country="{address/country}"
               city="{address/city}"
               street="{address/street}">
               <telephone number="{telephone}"/>
      </address>
</student>
   </xsl:template>
</xsl:stylesheet>
```

Для трансформации одного документа в другой можно использовать, например, следующий код.

```
/* пример # 14 : трансформация XML : SimpleTransform.java */
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class SimpleTransform {
   public static void main(String[] args) {
      try {
         TransformerFactory tf =
            TransformerFactory.newInstance();

         //установка используемого XSL-преобразования
         Transformer transformer =
tf.newTransformer(new StreamSource("students.xsl"));

         //установка исходного XML-документа и конечного XML-файла
         transformer.transform(
            new StreamSource("students.xml"),
               new StreamResult("newstudents.xml"));

            System.out.print("complete");
      } catch(TransformerException e) {
         e.printStackTrace();
      }
   }
}
```

В результате получится XML-документ **newstudents.xml** следующего вида:

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
   <student faculty="mmf">
      <name>Mitar Alex</name>
      <address country="Belarus" city="Minsk"
            street="Kalinovsky 45">
         <telephone number="3462356"/>
      </address>
   </student>
   <student faculty="mmf">
      <name>Pashkun Alex</name>
      <address country="Belarus" city="Brest"
            street="Knorina 56">
         <telephone number="4582356"/>
      </address>
   </student>
</students>
```

Элементы таблицы стилей

Таблица стилей представляет собой well-formed XML-документ. Эта таблица описывает изначальный документ, конечный документ и то, как трансформировать один документ в другой.

Какие же элементы используются в данном листинге?

```
<xsl:output method="xml" indent="yes"/>
```

Данная инструкция говорит о том, что конечный документ, который получится после преобразования, будет являться XML-документом.

```
<xsl:template match="student">
   <lastname>
      <xsl:apply-templates/>
   </lastname>
</xsl:template>
```

Инструкция **<xsl:template...>** задает шаблон преобразования. Набор шаблонов преобразования составляет основную часть таблицы стилей. В предыдущем примере приводится шаблон, который преобразует элемент **student** в элемент **lastname**.

Шаблон состоит из двух частей:

1. параметр **match**, который задает элемент или множество элементов в исходном дереве, где будет применяться данный шаблон;
2. содержимое шаблона, которое будет вставлено в конечный документ.

Нужно отметить, что содержимое параметра **math** может быть довольно сложным. В предыдущем примере просто ограничились именем элемента. Но, к примеру, следующее содержимое параметра **math** указывает на то, что шаблон должен применяться к элементу **url**, содержащему атрибут **protocol** со значением **mailto**:

```
<xsl:template match="url[@protocol='mailto']">
```

Кроме этого, существует набор функций, которые также могут использоваться при объявлении шаблона:

```
<xsl:template match="chapter[position()=2]">
```

Данный шаблон будет применен ко второму по счету элементу **chapter** исходного документа.

Инструкция **<xsl:apply-templates/>** сообщает XSL-процессору о том, что нужно перейти к просмотру дочерних элементов. Эта запись означает в расширенном виде:

```
<xsl:apply-templates select="child::node()" />
```

XSL-процессор работает по следующему алгоритму. После загрузки исходного XML-документа и таблицы стилей процессор просматривает весь документ от корня до листьев. На каждом шагу процессор пытается применить к данному элементу некоторый шаблон преобразования; если в таблице стилей для текущего просматриваемого элемента есть шаблон, процессор вставляет в результирующий документ содержимое этого шаблона. Когда процессор встречает инструкцию **<xsl:apply-templates/>**, он переходит к дочерним элементам текущего узла и повторяет процесс, т.е. пытается для каждого дочернего элемента найти соответствие в таблице стилей.

Задания к главе 16

Вариант А

Создать файл XML и соответствующее ему DTD-определение. Задать схему XSD. Определить класс Java, соответствующий данному описанию. Создать Java-приложение для инициализации массива объектов информацией из XML-файла. Произвести проверку XML-документа с привлечением DTD и XSD. Определить метод, производящий преобразование данного XML-документа в документ, указанный в задании.

1. **Оранжерея**.
   Растения, содержащиеся в оранжерее, имеют следующие характеристики:
   - Name – название растения.
   - Soil – почва для посадки, которая может быть следующих типов: подзолистая, грунтовая, дерново-подзолистая.
   - Origin – место происхождения растения.
   - Visual parameters (должно быть несколько) – внешние параметры: цвет стебля, цвет листьев, средний размер растения.
   - Growing tips (должно быть несколько) – предпочитаемые условия произрастания: температура (в градусах), освещение (светолюбиво либо нет), полив (мл в неделю).
   - Multiplying – размножение: листьями, черенками либо семенами.

   Корневой элемент назвать Flower.

   Создать XML файл, отображающий заданную тему, привести примеры 4-5 растений. С помощью XSL преобразовать данный файл в формат HTML, где отобразить растения по предпочитаемой температуре (по возрастанию).

2. **Алмазный фонд**.
   Драгоценные и полудрагоценные камни, содержащиеся в павильоне, имеют следующие характеристики:
   - Name – название камня.
   - Preciousness – может быть драгоценным либо полудрагоценным.
   - Origin – место добывания.
   - Visual parameters (должно быть несколько) – могут быть: цвет (зеленый, красный, желтый и т.д.), прозрачность (измеряется в процентах 0-100%), способы огранки (количество граней 4-15).
   - Value – вес камня (измеряется в каратах).

   Корневой элемент назвать Gem.

   Создать XML файл, отображающий заданную тему, привести примеры 4-5 камней. С помощью XSL преобразовать данный файл в формат XML , где корневым элементом будет место происхождения.
3. **Тарифы мобильных компаний**.
   Тарифы мобильных компаний могут иметь следующую структуру:
   - Name – название тарифа.
   - Operator name – название сотового оператора, которому принадлежит тариф.
   - Payroll – абонентская плата в месяц (0 – n рублей).
   - Call prices (должно быть несколько) – цены на звонки: внутри сети (0 – n рублей в минуту), вне сети (0 – n рублей в минуту), на стационарные телефоны (0 – n рублей в минуту).
   - SMS price – цена за смс (0 – n рублей).
   - Parameters (должно быть несколько) – наличие любимого номера (0 – n), тарификация (12-секундная, минутная), плата за подключение к тарифу (0 – n рублей).

   Корневой элемент назвать Tariff.

   Создать XML файл, отображающий заданную тему, привести примеры 4-5 тарифов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать тарифы по абонентской плате.
4. **Лекарственные препараты**.
   Лекарственные препараты имеют следующие характеристики.
   - Name – название препарата.
   - Price – цена за упаковку (0 – n рублей).
   - Dosage – дозировка препарата (мг/день).
   - Visual (должно быть несколько) – визуальные характеристики препарата: цвет (белый, желтый, зеленый, красный), консистенция (жидкий, порошкообразный, твердый), показания к применению (респираторные заболевания, расстройства организма, психические заболевания, общеукрепляющее).

   Корневой элемент назвать Medicine.

   Создать XML файл, отображающий заданную тему, привести примеры 4-5 лекарств. С помощью XSL преобразовать данный файл в формат HTML , при выводе отсортировать лекарства по цене.
5. **Компьютер**.
   Компьютерные комплектующие имеют следующие характеристики:

- Name – название комплектующего.
- Origin – страна производства.
- Price – цена (0 – n рублей).
- Type (должно быть несколько) – периферийное либо нет, энергопотребление (ватт), наличие кулера (есть либо нет), группа комплектующих (устройства ввода-вывода, мультимедийные), порты (COM, USB, LPT).
- Critical – критично ли наличие комплектующего для работы компьютера.

Корневой элемент назвать Device.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 устройств. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать Critical.

6. **Огнестрельное оружие**.

Огнестрельное оружие можно структурировать по следующей схеме:

- Model – название модели.
- Handy – одно- или двуручное.
- Origin – страна производства.
- TTC (должно быть несколько) – тактико-технические характеристики: дальнобойность (близкая [0 – 500м], средняя [500 – 1000 м], дальняя [1000 – n метров]), прицельная дальность (в метрах), наличие обоймы, наличие оптики.
- Material – материал изготовления.

Корневой элемент назвать Gun.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 видов. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать страну производства.

7. **Холодное оружие**.

Холодное оружие можно структурировать по следующей схеме:

- Type – тип (нож, кинжал, сабля и т.д.).
- Handy – одно или двуручное.
- Origin – страна производства.
- Visual (должно быть несколько) – визуальные характеристики: клинок (длина клинка [10 – n см], ширина клинка [10 – n мм]), материал (клинок [сталь, чугун, медь и т.д.]), рукоять (деревянная [если да, то указать тип дерева], пластик, металл), наличие кровостока (есть либо нет).
- Value – коллекционный либо нет.

Корневой элемент назвать Knife.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 видов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по длине клинка.

8. **Военные самолеты**.

Военные самолеты можно описать по следующей схеме:

- Model – название модели.
- Origin – страна производства.
- Chars (должно быть несколько) – характеристики, могут быть следующими: тип (самолет поддержки, сопровождения, истребитель, пере-

хватчик, разведчик), кол-во мест (1 либо 2), боекомплект (есть либо нет [разведчик], если есть, то: ракеты [0 – 10]), наличие радара.
- Parameters – длина (в метрах), ширина (в метрах), высота (в метрах).
- Price – цена (в долларах).

Корневой элемент назвать Plane.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов самолетов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по стоимости.

9. **Конфеты**.
   - Name – название конфеты.
   - Energy– калорийность (ккал).
   - Type (должно быть несколько) – тип конфеты (карамель, ирис, шоколадная [с начинкой либо нет]).
   - Ingredients (должно быть несколько) – ингредиенты: вода, сахар (в мг), фруктоза (в мг), тип шоколада (для шоколадных), ванилин (в мг)
   - Value – пищевая ценность: белки (в гр.), жиры (в гр.) и углеводы (в гр.).
   - Production – предприятие-изготовитель.

   Корневой элемент назвать Candy.

   Создать XML файл, отображающий заданную тему, привести примеры 4-5 конфет. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по месту изготовления.
10. **Пиво**.
    - Name – название пива.
    - Type – тип пива (темное, светлое, лагерное, живое).
    - Al – алкогольное либо нет.
    - Manufacturer – фирма-производитель.
    - Ingredients (должно быть несколько) – ингредиенты: вода, солод, хмель, сахар и т.д.
    - Chars (должно быть несколько) – характеристики: кол-во оборотов (если алкогольное), прозрачность (в процентах), фильтрованное либо нет, пищевая ценность (ккал), способ разлива (объем и материал емкостей)
    - Корневой элемент назвать Beer.

      Создать XML-файл, отображающий заданную тему, привести примеры 4-5 сортов пива. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать производителя.
11. **Периодические издания**.
    - Title – название издания.
    - Type – тип издания (газета, журнал, буклет).
    - Monthly – ежемесячное либо нет.
    - Chars (должно быть несколько) – характеристики: цветное (да либо нет), объем (n страниц), глянцевое (да [только для журналов и буклетов] либо нет [для газет]), имеет подписной индекс (только для газет и журналов).

    Корневой элемент назвать Paper.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов периодики. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать тип (Type).

12. **Интернет-страницы**.
- Title – название страницы.
- Type – тип страницы (рекламный, страница новостей, портал, зеркало).
- Chars (должно быть несколько) – наличие электронного ящика (только для порталов, зеркал и страниц новостей), наличие новостей (только для страниц новостей), наличие архивов для выкачивания (только для зеркал), наличие голосования (есть[если есть, то анонимное либо с применением авторизации] либо нет), платный (информация, доступная для выкачивания, бесплатна либо нет).
- Authorize – необходима либо нет авторизация.

Корневой элемент назвать Site.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов периодики. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать тип (Type).

Тестовые задания к главе 16

Вопрос 16.1.

Какой существует способ описания данных в XML? (выберите два)

1. XML использует DTD для описания данных
2. XML использует XSL для описания данных
3. XML использует XSD для описания данных
4. XML использует CSS для описания данных

Вопрос 16.2.

В каких строках XML документа есть ошибки? (выберите два)

```
1 <?xml version="1.0"?>
2 <folder>
3  <file><name><contents></contents></name></file>
4  <file><name/><contents></contents><name/></file>
5  <file><name/><contents></contents></name></file>
6  <file><name><contents/><name/></file>
7 </folder>
```

1. 1;
2. 2;
3. 3;
4. 4;
5. 5;
6. 6;
7. 7;
8. нет ошибок.

Вопрос 16.3.

Какое из данных имен не является корректным именем для XML элемента? (выберите 2)

1. <hello_dolly>;
2. <big bang>;
3. <xmldocument>;
4. <7up>;
5. только одно имя некорректно.

Вопрос 16.4.

Значения атрибутов XML всегда должны помещаться в …? (выберите два)

1. двойные кавычки “ ”;
2. апострофы ‘ ‘;
3. фигурные скобки { };
4. квадратные скобки [];
5. могут обходиться без ограничивающих символов.

Вопрос 16.5.

Какие виды событий нельзя обрабатывать с помощью SAX-анализатора?

1. события документа;
2. события загрузки DTD-описаний;
3. события при анализе DTD-описаний;
4. ошибки;
5. все перечисленные события можно обработать.