Глава 14

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 хранящиеся в этом формате данные и быстро производить базовые операции над этими данными.

XML является упрощенным 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 — текстовое представление информации 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 названия книги, его можно создать:

```
<title>Java SE 8</title>
<title book="Java SE 8"/>
<title-book>Java SE 8</title-book>
```

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

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

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 все, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы.

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

```
<book date-of-issue="04/11/2011" title='Java Industrial'/>
```

В отличие от этого HTML разрешает записывать значение атрибута без кавычек.

Например:

```
<FORM method=POST action=index.jsp>
```

Далее представлены примеры неправильной орфографии и использования тегов:

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

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>.

```
<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.

Любой открывающий тег должен иметь закрывающий.

```
<book>
          <system-exit>Zukov
</book>
```

Если тег не имеет содержимого, можно использовать конструкцию вида **<system-exit/>**. В HTML есть возможность не закрывать теги, и браузер определяет стили по открывающемуся тегу.

Наименования тегов чувствительны к регистру (case-sensitive), т. е., например, теги **<author>**, **<Author>**, **<AUTHOR>** будут совершенно разными. При работе с XML-тегом вида **<system-exit>Zukov</System-Exit>** программа-анализатор просто не найдет завершающий тег и выдаст ошибку. Язык HTML нетребователен к регистру.

Пусть существует XML-документ **students.xml** с данными о студентах:

 # 1 # описание студентов # students.xml

```
<?xml version="1.0" encoding="UTF-8"?>
        <students>
                <student login="MitarAlex7" faculty="mmf">
                        <name>Mitar Alex</name>
                        <telephone>2456474</telephone>
                        <address>
                                <country>Belarus</country>
                                <city>Minsk</city>
                                <street>Kalinovsky 45</street>
                        </address>
                </student>
                <student login="Pashkin5" faculty="mmf">
                        <name>Pashkin Alex</name>
                        <telephone>3453789</telephone>
                        <address>
                                <country>Belarus</country>
                                <city>Brest</city>
                                <street>Knorina 56</street>
                        </address>
                </student>
        </students>
```

Документ обладает древовидной структурой, следовательно, в базе данных по этому описанию требовалось бы создать две таблицы.

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 заключена между **<?** и **?>** и начинаться с идентификатора, называемого **target** (цель).

Например:

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

Эта инструкция по обработке сообщает браузеру, что для данного документа необходимо применить стилевую таблицу (stylesheet) **student.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 --> -->
```

не внутри значений атрибутов:

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

не внутри тегов:

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

Указатели

Текстовые блоки XML-документа не могут содержать символы, 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>
```

Корректность

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, определенных 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

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>
```

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 способна 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 «+» (1 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:

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 'Oracle'>
<sender>&company;</sender>
```

Программа-анализатор, которая будет обрабатывать файл, автоматически подставит значение Oracle 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:

 # 2 # dtd для документа students.xml # students.dtd

```
<?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)>
```

Одна из причин отказа от DTD-описаний — его представление в виде документа, не являющегося по определению XML-документом.

Схема XSD

Схема XSD представляет собой более строгое, чем DTD, руководство по созданию и валидации XML-документа. XSD-схема, в отличие от DTD, является XML-документом, и поэтому она отличается гибкостью при использовании в приложениях, при задании правил документа, а также для дальнейшего расширения новой функциональностью. В отличие от DTD схема содержит большое количество базовых типов (44 типа) и имеет поддержку пространств имен (**namespace**). С помощью схемы XSD можно также проверить документ на корректность, а именно валидность.

Схема XSD первой строкой содержит XML-декларацию. Любая схема своим корневым элементом должна содержать элемент **schema**.

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

name — определяет имя элемента;

type — указывает тип элемента;

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

minOccurs и **maxOccurs** — количество повторений этого элемента (по умолчанию принимает значение 1), чтобы указать, что количество элементов не ограничено, в атрибуте **maxOccurs** необходимо задать **unbounded**.

```
<element name="telephone" type="positiveInteger" />
```

или

```
<element name="student"
          type="tns:Student"
          minOccurs="1"
          maxOccurs="unbounded" />
```

Если стандартных типов не хватает для полноты описания элемента, то 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** (уникальный идентификатор), **gYear** (год) 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, расположенный 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, задаваемый регулярным выражением;

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="(\\w){8, 20}"/>
          </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 внутри описываемого сложного типа должны располагаться в определенной последовательности, используются элементы **sequence**, **all**, **choice**. 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="positiveInteger"/>
                <element name="address" type="tns:Address"/>
        </sequence>
        <attribute name="login" type="tns:Login" use="required"/>
        <attribute name="faculty" type="string" use="optional"/>
</complexType>
```

Для задания произвольного порядка следования элементов в XML используется такой тег, как **<all>**, который допускает любой порядок.

```
<element name="employee">
        <complexType>
                <all>
                        <element name="phone" type="positiveInteger"/>
                        <element name="salary" type="decimal"/>
                </all>
        </complexType>
</element>
```

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

Если набор значений поля или атрибута ограничен некоторым множеством, то для его определения следует использовать элемент **enumeration**. Например, атрибут **faculty** может принимать только значения: **mmf**, **geo**, **ksis**. Тогда вместо элемента

```
<attribute name="faculty" type="string" use="optional" />
```

следует записать

```
<attribute name="faculty">
          <simpleType>
                    <restriction base="string">
                              <enumeration value="mmf"></enumeration>
                              <enumeration value="geo"></enumeration>
                              <enumeration value="ksis"></enumeration>
                    </restriction>
          </simpleType>
</attribute>
```

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

 # 3 # xsd-схема для документа students.xml # students.xsd

```
<?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"
          elementFormDefault="qualified">
          <element name="students">
                    <complexType>
                              <sequence>
                                        <element name="student"
                                                  type="tns:Student"
                                                  minOccurs="2"
                                                  maxOccurs="unbounded" />
                              </sequence>
                    </complexType>
          </element>
          <complexType name="Student">
                    <sequence>
                              <element name="name" type="string" />
                              <element name="telephone" type="positiveInteger" />
                              <element name="address" type="tns:Address" />
                    </sequence>
          <attribute name="login" type="tns:Login" use="required" />
          <attribute name="faculty" use="optional" default="mmf">
                    <simpleType>
                              <restriction base="string">
                                        <enumeration value="mmf"></enumeration>
                                        <enumeration value="geo"></enumeration>
                                        <enumeration value="ksis"></enumeration>
                              </restriction>
                    </simpleType>
          </attribute>
          </complexType>
          <simpleType name="Login">
                    <restriction base="ID">
                              <pattern value="([a-zA-Z])[a-zA-Z0-9]{7,19}"/>
```

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

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

```
<element name="Teacher">
        <complexType>
                <simpleContent>
                        <extension base="string">
                                <attribute name="faculty" type="string"/>
                        </extension>
                </simpleContent>
        </complexType>
</element>
```

Для расширения/ограничения ранее объявленных сложных типов используется элемент **complexContent**. Пусть имеется некоторый тип **PersonType**, содержащий элементы **name**, **telephone** и **address**. Типы **Student** и **Abiturient** не только содержат те же элементы, что и **PersonType**, но и добавляют к ним свои. Тип **Student** добавляет к тегу **student** атрибуты **login** и **faculty**, а тип **Abiturient** добавляет элемент **average-mark**. Тип **PersonType** выглядит следующим образом:

```
<complexType name="PersonType">
        <sequence>
                <element name="name" type="string" />
                <element name="telephone" type="positiveInteger" />
                <element name="address" type="tns:Address" />
        </sequence>
</complexType>
```

Типы **Student** и **Abiturient**, его расширяющие, представлены в виде:

```
<complexType name="Student">
        <complexContent>
                <extension base="tns:PersonType">
                        <attribute name="login" type="tns:Login" use="required" />
                        <attribute name="faculty" use="optional" default="mmf">
                                <simpleType>
                                        <restriction base="string">
                                        <enumeration value="mmf"></enumeration>
```

```
                                        <enumeration value="geo"></enumeration>
                                        <enumeration value="ksis"></enumeration>
                                        </restriction>
                                </simpleType>
                        </attribute>
                </extension>
        </complexContent>
</complexType>
<complexType name="Abiturient">
        <complexContent>
                <extension base="tns:PersonType">
                        <sequence>
                                <element name="average-mark" type="double" />
                        </sequence>
                </extension>
        </complexContent>
</complexType>
```

Соответствие типов и тегов записывается в виде:

```
<element name="person" type="tns:PersonType" abstract="true"></element>
<element name="student" type="tns:Student" substitutionGroup="tns:person"></element>
<element name="abiturient" type="tns:Abiturient"  substitutionGroup="tns:person"></element>
```

Корневой элемент будет ссылаться только на абстрактный элемент **person**.

```
<element name="students">
        <complexType>
                <sequence>
                        <element ref="tns:person" minOccurs="2" maxOccurs="unbounded" />
                </sequence>
        </complexType>
</element>
```

Документ, соответствующий описанным выше правилам выглядит следующим образом:

 # 4 # документ с описанием студентов и абитуриентов # students_ext.xml

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.example.com/students"
        xsi:schemaLocation="http://www.example.com/students person.xsd">
        <abiturient>
                <name>Petkevich</name>
                <telephone>2787474</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Minsk</city>
                        <street>Slobodskaja 7</street>
                </address>
                <average-mark>9.0</average-mark>
        </abiturient>
```

```
        <student login="MitarAlex7" faculty="mmf">
                <name>Mitar Alex</name>
                <telephone>2456474</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Minsk</city>
                        <street>Kalinovsky 45</street>
                </address>
        </student>
        <student login="Pashkin5" faculty="mmf">
                <name>Pashkin Alex</name>
                <telephone>3453789</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Brest</city>
                        <street>Knorina 56</street>
                </address>
        </student>
</students>
```

В приведенном документе 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 имен распространяется на элемент, где он объявлен, и на все дочерние элементы.

Тогда для проверки документа экземпляру-парсеру следует дать указание использовать DTD или схему XSD. В XML-документ для валидации с помощью схемы следует добавить вместо корневого элемента **<students>** элемент **<ns:students>** вида:

```
<ns:students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:ns="http://www.example.com/students"
        xsi:schemaLocation="http://www.example.com/students students.xsd">
```

или

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

Следующий пример выполняет проверку документа на валидность, то есть соответствие схеме, средствами языка Java при парсинге документа.

/ # 5 # проверка корректности документа XML с XSD # ValidatorSAX.java */*

```
package by.bsu.valid;
import java.io.File;
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
public class ValidatorSAX {
          public static void main(String[ ] args) {
                    String filename = "data/students.xml";
                    String schemaname = "data/students.xsd";
                    String logname = "logs/log.txt";
                    Schema schema = null;
                    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
                    SchemaFactory factory = SchemaFactory.newInstance(language);
                    try {
                              // установка проверки с использованием XSD
                              schema = factory.newSchema(new File(schemaname));
                              SAXParserFactory spf = SAXParserFactory.newInstance();
                              spf.setSchema(schema);
                              // создание объекта-парсера
                              SAXParser parser = spf.newSAXParser();
                              // установка обработчика ошибок и запуск
                              parser.parse(filename, new StudentErrorHandler(logname));
                              System.out.println(filename + " is valid");
                    } catch (ParserConfigurationException e) {
                              System.err.println(filename + " config error: " + e.getMessage());
                    } catch (SAXException e) {
                              System.err.println(filename + " SAX error: " + e.getMessage());
                    } catch (IOException e) {
                              System.err.println("I/O error: " + e.getMessage());
                    }
          }
}
```

Для запуска этого примера используются только стандартные библиотеки JDK.

Класс обработчика ошибок может выглядеть следующим образом:

/ # 6 # обработчик ошибок # StudentErrorHandler.java */*

```
package by.bsu.valid;
import java.io.IOException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
public class StudentErrorHandler extends DefaultHandler {
                    // создание регистратора ошибок для пакета by.bsu.valid
          private Logger logger = Logger.getLogger("by.bsu.valid");
          public StudentErrorHandler(String log) throws IOException {
                    // 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();
          }
}
```

При проверке XML-документа разумно все ошибки фиксировать, в частно-сти, с помощью логгера Log4J, библиотеку которого log4j-[*version*].jar следует подключить к проекту.

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 'MitarAlex7'.
ERROR - 14 : 41 - cvc-attribute.3: The value 'MitarAlex7' 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 угловую скобку в элементе **telephone**, будет выведено сообщение о фатальной ошибке:

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

Можно также провести проверку документа на соответствие XSD с применением возможностей специального класса **Validator**.

/ # 7 # проверка корректности документа XML # ValidatorSAX.java */*

```
package by.bsu.valid;
import java.io.*;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class ValidatorSAXXSD {
    public static void main(String[ ] args) {
         String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
         String fileName = "data/students.xml";
         String schemaName = "data/students.xsd";
         SchemaFactory factory = SchemaFactory.newInstance(language);
         File schemaLocation = new File(schemaName);
         try {
              // создание схемы
              Schema schema = factory.newSchema(schemaLocation);
              // создание валидатора на основе схемы
              Validator validator = schema.newValidator();
              // проверка документа
              Source source = new StreamSource(fileName);
              validator.validate(source);
              System.out.println(filename + " is valid.");
         } catch (SAXException e) {
              System.err.print("validation "+ filename + " is not valid because "
                + e.getMessage());
         } catch (IOException e) {
              System.err.print(filename + " is not valid because "
              + e.getMessage());
         }
    }
}
```

Экземпляр класса **Validator** может использовать класс-обработчик ошибок с сохранением информации о них в файле **log.txt**. В этом случае в код примера следует вставить следующий фрагмент, осуществляющий инициализацию и установку объекта **StudentErrorHandler** для экземпляра **Validator**.

```
StudentErrorHandler sh = new StudentErrorHandler("logs/log.txt");
validator.setErrorHandler(sh);
validator.validate(source);
System.out.println(filename + " validating is ended.");
```

Информация о некорректном содержимом XML-файла в этом случае сохраняется в log-файле и на консоль выводиться не будет.

JAXB. Маршаллизация и демаршаллизация

Начиная с версии Java 6, включены продвинутые механизмы извлечения/сохранения данных при взаимодействии с XML.

Маршаллизация — механизм преобразования данных из java-объектов в конкретное хранилище, будь то документ XML, база данных или простой текстовый файл.

Демаршаллизация — обратный процесс преобразования данных из внешних источников в структуру хранения, поддерживаемую виртуальной машиной. Проблемой остается организация взаимно однозначного соответствия информации в источнике, например, XML-документе, и экземпляре типа данных, принимающем эту информацию.

Соединение этих двух процессов должно корректно определять импорт-экспорт или круговорот информации без потерь и искажений на всех этапах. Информация, взятая из XML-файла и транслированная в объект java, при обратном преобразовании должна быть возвращена в идентичном виде.

Следующий пример на основе экземпляра класса **Students,** содержащего, в свою очередь, список экземпляров класса **Student**, создает структуру документа XML и сохраняет в ней информацию из объекта. Классы **Students** и **Student** разработаны и аннотированы так, чтобы структура создаваемого на их основе XML-документа соответствовала документу **students.xml**, приведенному в листинге #1.

/ # 8 # компонент JavaBean # Student.java */*

```
package by.bsu.xmlstudents;
import javax.xml.bind.annotation.*;
import javax.xml.bind.annotation.adapters.*;
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Student", propOrder = {
    "name",
    "telephone",
    "address"
}) // задание последовательности элементов XML
public class Student {
          @XmlAttribute(required = true)
@XmlJavaTypeAdapter(CollapsedStringAdapter.class)
          @XmlID
          private String login;
          @XmlElement(required = true)
          private String name;
```

```
@XmlAttribute(required = false)
private String faculty;
@XmlElement(required = true)
private int telephone;
@XmlElement(required = true)
private Address address = new Address();
public Student() { } // необходим для маршаллизации/демаршалиизации XML
public Student(String login, String name, String faculty, int telephone, Address address) {
          this.login = login;
          this.name = name;
          this.faculty = faculty;
          this.telephone = telephone;
          this.address = 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 int getTelephone() {
          return telephone;
}
public void setTelephone(int telephone) {
          this.telephone = telephone;
}
public Address getAddress() {
          return address;
}
public void setAddress(Address address) {
          this.address = address;
}
public String toString() {
          return "\nLogin: " + login + "\nName: " + name + "\nTelephone: " + telephone
                    + "\nFaculty: " + faculty + address.toString();
}
@XmlRootElement
@XmlType(name = " address ", propOrder = {
```

```
                                "city",
                                "country",
                                "street"
        })
        public static class Address { // внутренний класс
                private String country;
                private String city;
                private String street;
                public Address() {// необходим для маршаллизации/демаршалиизации XML
                }
                public Address(String country, String city, String street) {
                        this.country = country;
                        this.city = city;
                        this.street = 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;
                }
                public String toString() {
                        return "\nAddress:" + "\n\tCountry: " + country
                        + "\n\tCity: " + city + "\n\tStreet: " + street + "\n";
                }
        }
}
```

Структура класса **Students** предназначена для хранения экземпляров класса **Student**. Класс **Student** в дальнейшем будет использоваться для создания списков и множеств объектов при разработке парсеров на основе информации, извлеченной из XML-документа.

/ # 9 # компонент для хранения списка студентов # Students.java */*

```
package by.bsu.xmlstudents;
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
```

```
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Students {
          @XmlElement(name="student")
          private ArrayList<Student> list = new ArrayList<Student>();
          public Students() {
                    super();
          }
          public void setList(ArrayList<Student> list) {
                    this.list = list;
          }
          public boolean add(Student st) {
                    return list.add(st);
          }
          @Override
          public String toString() {
                    return "Students [list=" + list + "]";
          }
}
```

Процесс маршаллизации состоит из создания JAXB контекста на основе класса **Students**, создания на его основе экземпляра типа **Marshaller** и сохранения информации в файл.

/ # 10 # создание XML-документа на основе экземпляра класса # MarshalMain.java */*

```
package by.bsu.jaxb;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import by.bsu.xmlstudents.Student;
import by.bsu.xmlstudents.Students;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
public class MarshalMain {
    public static void main(String[] args) {
        try {
          JAXBContext context = JAXBContext.newInstance(Students.class);
          Marshaller m = context.createMarshaller();
          Students st = new Students() { // анонимный класс
            {
            // добавление первого студента
            Student.Address addr = new Student.Address("BLR", "Minsk", "Skoriny 4");
            Student s = new Student("gochette", "Klimenko", "mmf", 2095306, addr);
            this.add(s);
            // добавление второго студента
            addr = new Student.Address("BLR", "Polotesk", "Simeona P. 23");
            s = new Student("ivette", "Teran", "mmf", 2345386, addr);
            this.add(s);
            }
```

```
        };
        m.marshal(st, new FileOutputStream("data/studs_marsh.xml"));
        m.marshal(st, System.out); // копия на консоль
        System.out.println("XML-файл создан");
      } catch (FileNotFoundException e) {
        System.out.println("XML-файл не может быть создан: " + e);
      } catch (JAXBException e) {
        System.out.println("JAXB-контекст ошибочен " + e);
      }
   }
}
```

В результате компиляции и запуска программы будет создан XML-документ

 # 11 # сохраненная информация # studs_marsh.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<students>
          <student login="gochette" faculty="mmf">
                    <name>Klimenko</name>
                    <telephone>2095306</telephone>
                    <address>
                              <city>Minsk</city>
                              <country>BLR</country>
                              <street>Skoriny 4</street>
                    </address>
          </student>
          <student login="ivette" faculty="mmf">
                    <name>Teran</name>
                    <telephone>2345386</telephone>
                    <address>
                              <city>Polotesk</city>
                              <country>BLR</country>
                              <street>Simeona P. 23</street>
                    </address>
          </student>
</students>
```

Процедура демаршаллизации аналогична маршаллизации с той разницей, что в итоге будет получен корректно созданный экземпляр класса **Students**.

/ # 12 # создание экземпляра класса на основе XML-документа # UnMarshalMain.java */*

```
package by.bsu.jaxb;
import java.io.FileNotFoundException;
import java.io.FileReader;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import by.bsu.xmlstudents.Students;
```

```
public class UnMarshalMain {
          public static void main(String[ ] args) {
                    try {
                              JAXBContext jc = JAXBContext.newInstance(Students.class);
                              Unmarshaller u = jc.createUnmarshaller();
                              FileReader reader = new FileReader("data/studs_marsh.xml");
                              Students students = (Students) u.unmarshal(reader);
                              System.out.println(students);
                    } catch (JAXBException e) {
                              e.printStackTrace();
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
          }
}
```

Вывод на консоль показывает идентичность информации после восстановления экземпляра.

Students [list=[
Login: gochette
Name: Klimenko
Telephone: 2095306
Faculty: mmf
Address:
Country: BLR
City: Minsk
Street: Skoriny 4
,
Login: ivette
Name: Teran
Telephone: 2345386
Faculty: mmf
Address:
Country: BLR
City: Polotesk
Street: Simeona P. 23
]]

JAXB. Генерация классов

Возможен инжиниринг классов на языке Java на основе XML-схемы. Ниже приведена схема **person.xsd** с описанием типов-классов **PersonType** и его расширений-подклассов **Student** и **Abiturient**, а также типа-класса **Students**, который может содержать список студентов и абитуриентов.

 # 13 # схема с описанием иерархии # person.xsd

```
<?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" elementFormDefault="qualified">
          <element name="person" type="tns:PersonType" abstract="true"></element>
          <element name="student" type="tns:Student" substitutionGroup="tns:person"></element>
          <element name="abiturient" type="tns:Abiturient"
                    substitutionGroup="tns:person"></element>
          <element name="students">
                    <complexType>
                              <sequence>
                                        <element ref="tns:person" minOccurs="2"
                                                                  maxOccurs="unbounded" />
                              </sequence>
                    </complexType>
          </element>
          <complexType name="PersonType">
                    <sequence>
                              <element name="name" type="string" />
                              <element name="telephone" type="positiveInteger" />
                              <element name="address" type="tns:Address" />
                    </sequence>
          </complexType>
          <complexType name="Student">
                    <complexContent>
                              <extension base="tns:PersonType">
                                        <attribute name="login" type="tns:Login" use="required" />
                                        <attribute name="faculty" use="optional" default="mmf">
                                                  <simpleType>
                                                            <restriction base="string">
                                                  <enumeration value="mmf"></enumeration>
                                                  <enumeration value="geo"></enumeration>
                                                  <enumeration value="ksis"></enumeration>
                                                            </restriction>
                                                  </simpleType>
                                        </attribute>
                              </extension>
                    </complexContent>
          </complexType>
          <complexType name="Abiturient">
                    <complexContent>
                              <extension base="tns:PersonType">
                                        <sequence>
                                                  <element name="average-mark" type="double" />
                                        </sequence>
                              </extension>
                    </complexContent>
```

```
        </complexType>
        <simpleType name="Login">
                <restriction base="ID">
                        <pattern value="([a-zA-Z])[a-zA-Z0-9]{7,19}" />
                </restriction>
        </simpleType>
        <complexType name="Address">
                <sequence>
                        <element name="country" type="string" />
                        <element name="city" type="string" />
                        <element name="street" type="string" />
                </sequence>
        </complexType>
</schema>
```

Запуск процесса генерации осуществляется с помощью командной строки:

```
xjc.exe person.xsd
```

В результате будет сгенерирован пакет **com.example.students**, содержащий следующие классы-сущности:

/ # 14 # исходный код классов, сгенерированный на основе XSD # PersonType.java # Abiturient.java # Student.java # Address.java # Students.java */*

```
package com.example.students;
import java.math.BigInteger;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlSchemaType;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PersonType", propOrder = {
    "name",
    "telephone",
    "address"
})
@XmlSeeAlso({
    Student.class,
    Abiturient.class
})
public class PersonType {
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    @XmlSchemaType(name = "positiveInteger")
    protected BigInteger telephone;
    @XmlElement(required = true)
    protected Address address;
```

```
    public String getName() {
        return name;
    }
    public void setName(String value) {
        this.name = value;
    }
    public BigInteger getTelephone() {
        return telephone;
    }
    public void setTelephone(BigInteger value) {
        this.telephone = value;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address value) {
        this.address = value;
    }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Abiturient", propOrder = { "averageMark" })
public class Abiturient extends PersonType {
          @XmlElement(name = "average-mark")
          protected double averageMark;
          public double getAverageMark() {
                    return averageMark;
          }
          public void setAverageMark(double value) {
                    this.averageMark = value;
          }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlID;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.CollapsedStringAdapter;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Student")
public class Student extends PersonType {
    @XmlAttribute(name = "login", required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    @XmlID
```

```
    protected String login;
    @XmlAttribute(name = "faculty")
    protected String faculty;
    public String getLogin() {
        return login;
    }
    public void setLogin(String value) {
        this.login = value;
    }
    public String getFaculty() {
        if (faculty == null) {
            return "mmf";
        } else {
            return faculty;
        }
    }
    public void setFaculty(String value) {
        this.faculty = value;
    }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Address", propOrder = {
    "country",
    "city",
    "street"
})
public class Address {
    @XmlElement(required = true)
    protected String country;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String street;
    public String getCountry() {
        return country;
    }
    public void setCountry(String value) {
        this.country = value;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String value) {
        this.city = value;
    }
    public String getStreet() {
```

```
        return street;
    }
    public void setStreet(String value) {
        this.street = value;
    }
}
package com.example.students;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElementRef;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "person"
})
@XmlRootElement(name = "students")
public class Students {
    @XmlElementRef(name = "person", namespace = "http://www.example.com/students", type =
JAXBElement.class)
    protected List<JAXBElement<? extends PersonType>> person;
    public List<JAXBElement<? extends PersonType>> getPerson() {
        if (person == null) {
            person = new ArrayList<JAXBElement<? extends PersonType>>();
        }
        return this.person;
    }
}
```

Также сгенерирован класс-фабрика для создания экземпляров перечисленных классов:

/ # 15 # исходный код классов, сгенерированный на основе XSD # ObjectFactory.java */*

```
package com.example.students;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;
@XmlRegistry
public class ObjectFactory {
    private final static QName _Person_QNAME =
                    new QName("http://www.example.com/students", "person");
    private final static QName _Student_QNAME =
                    new QName("http://www.example.com/students", "student");
    private final static QName _Abiturient_QNAME =
                    new QName("http://www.example.com/students", "abiturient");
```

```
    public ObjectFactory() {
    }
    public Students createStudents() {
        return new Students();
    }
    public PersonType createPersonType() {
        return new PersonType();
    }
    public Student createStudent() {
        return new Student();
    }
    public Abiturient createAbiturient() {
        return new Abiturient();
    }
    public Address createAddress() {
        return new Address();
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "person")
    public JAXBElement<PersonType> createPerson(PersonType value) {
        return new JAXBElement<PersonType>(_Person_QNAME, PersonType.class, null, value);
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "student",
                            substitutionHeadNamespace = "http://www.example.com/students",
                            substitutionHeadName = "person")
    public JAXBElement<Student> createStudent(Student value) {
        return new JAXBElement<Student>(_Student_QNAME, Student.class, null, value);
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "abiturient",
                            substitutionHeadNamespace = "http://www.example.com/students",
                            substitutionHeadName = "person")
    public JAXBElement<Abiturient> createAbiturient(Abiturient value) {
        return new JAXBElement<Abiturient>(_Abiturient_QNAME, Abiturient.class, null, value);
    }
}
```

Демаршаллизацию информации из XML-документа для более высокого качества следует производить с применением XSD-схемы:

/ # 16 # создание экземпляра класса из XML с привлечением XSD # UnMarshalWithXSD.java */*

```
package by.bsu.jaxb;
import java.io.File;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
```

```
import com.example.students.Students;
public class UnMarshalWithXSD {
    public static void main(String[ ] args) {
        JAXBContext jc = null;
        try {
                  jc = JAXBContext.newInstance("com.example.students");
                  Unmarshaller um = jc.createUnmarshaller();
                  String schemaName = "dat/person.xsd";
                  SchemaFactory factory =
        SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
                  File schemaLocation = new File(schemaName);
                  // создание схемы и передача ее демарашаллизатору
                  Schema schema = factory.newSchema(schemaLocation);
                  um.setSchema(schema);
                  Students st =
                        (Students) um.unmarshal(new File("data/students_ext.xml"));
                  System.out.println(st);
        } catch (JAXBException e) {
                    printStackTrace();
        } catch (SAXException e) {
                    printStackTrace();
        }
    }
}
```

JAXP

JAXP — Java API for XML Processing. XML-документ 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 проекте. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки — XML-анализаторы (парсеры).

Парсер — это библиотека, которая читает XML-документ, а затем предоставляет набор методов для обработки информации из этого документа.

Валидирующие и невалидирующие анализаторы

Как было уже упомянуто, существуют 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 XSD.

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.

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

Древовидная и псевдособытийная 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 прохождении 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 квазисобытия, — это SAX-анализаторы.

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

В первом случае анализатор строит в памяти объект, представляющий собой дерево из элементов, соответствующее XML-документу. Далее вся работа ведется именно с этим объектом-деревом.

Во втором случае анализатор работает следующим образом: при чтении/анализе 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, анализатор 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.

Анализатор StAX работает подобно итератору, 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-документа 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.

Псевдособытийная модель

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует квазисобытие при нахождении элемента, атрибута или текста. 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-документе.

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

SAX-анализаторы

SAX API определяет ряд интерфейсов, используемых при разборе документа. Чаще других используется **org.xml.sax.ContentHandler** и некоторые объявленные в нем методы:

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). Если этой информации достаточно много, то метод может быть вызван более одного раза.

Для обработки предупреждений и ошибок, возникающих при разборе XML-документа, применяется интерфейс **org.xml.sax.ErrorHandler**, содержащий методы:

```
warning(SAXParseException e),
error(SAXParseException e),
fatalError(SAXParseException e).
```

В пакете **org.xml.sax** в SAX2 API содержатся также интерфейсы **DTDHandler**, **DocumentHandler** и **EntityResolver**, которые необходимо реализовать для обработки интересующего 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**, **DocumentHandler**) или наследует класс **org.xml.sax.helpers.DefaultHandler**, и реализовать методы, отвечающие за обработку интересующих частей документа или ошибок.
2. Используя SAX2 API, поддерживаемое всеми SAX-парсерами, создать **org.xml.sax.XMLReader**, например:

```
XMLReader reader = XMLReaderFactory.createXMLReader();
```

или

```
XMLReader reader =
                XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
```

Для библиотеки **xercesImpl.jar**, которую можно загрузить по адресу **http://xerces.apache.org/xerces2-j/**.

3. Передать в **XMLReader** объект класса, созданного на шаге 1 с помощью соответствующих методов: **setContentHandler()**, **setErrorHandler()**, **setDTDHandler()**, **setEntityResolver()**.
4. Вызвать метод **parse(String filename)** класса **XMLReader**, которому в качестве параметров передать путь (URI) к анализируемому документу либо **InputSource**.

Следующий пример в результате парсинга выводит на консоль содержимое XML-документа.

/ # 17 # чтение и вывод XML-документа # SimpleStudentHandler.java */*

```
package by.bsy.saxsimple;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
public class SimpleStudentHandler extends DefaultHandler {
          @Override
          public void startDocument() {
                    System.out.println("Parsing started");
          }
          @Override
          public void startElement(String uri, String localName, String qName, Attributes attrs) {
                    String s = localName;
                    // получение и вывод информации об атрибутах элемента
                    for (int i = 0; i < attrs.getLength(); i++) {
                              s += " " + attrs.getLocalName(i) + "=" + attrs.getValue(i);
```

```
                }
                System.out.print(s.trim());
        }
        @Override
        public void characters(char[ ] ch, int start, int length) {
                System.out.print(new String(ch, start, length));
        }
        @Override
        public void endElement(String uri, String localName, String qName) {
                System.out.print(localName);
        }
        @Override
        public void endDocument() {
                System.out.println("\nParsing ended");
        }
}
```

Где **uri** — уникальное название **namespace**, **localName** — имя элемента без префикса, задаваемого именем атрибута **xmlns**, например:

```
xmlns:ns=http://www.example.com/Students
```

то есть без **ns**, **qName** — полное имя элемента с префиксом, **attrs** — список атрибутов.

/ # 18 # создание и запуск простейшего парсера # SAXSimpleMain.java*/*

```
package by.bsy.saxsimple;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.SAXException;
import java.io.IOException;
public class SAXSimpleMain {
        public static void main(String[ ] args) {
                try {
                        // создание SAX-анализатора
                        XMLReader reader = XMLReaderFactory.createXMLReader();
                        SimpleStudentHandler handler = new SimpleStudentHandler();
                        reader.setContentHandler(handler);
                        reader.parse("data/students.xml");
                } catch (SAXException e) {
                        System.err.print("ошибка SAX парсера " + e);
                } catch (IOException e) {
                        System.err.print("ошибка I/O потока " + e);
                }
        }
}
```

В результате в консоль будет выведено (в XML-документе должна быть ссылка на XSD):

```
Parsing started
students xsi:schemaLocation=http://www.example.org/Students.xsd students.xsd
   student login=MitarAlex7 faculty=mmf
      nameMitar Alexname
      telephone2456474telephone
      address
         countryBelaruscountry
         cityMinskcity
         streetKalinovsky 45street
      address
   student
   student login=Pashkin5 faculty=mmf
      namePashkin Alexname
      telephone3453789telephone
      address
         countryBelaruscountry
         cityBrestcity
         streetKnorina 56street
      address
   student
students
Parsing ended
```

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

/ # 19 # формирование коллекции объектов на основании разбора XML-документа # StudentsSAXBuilder.java */*

```
package by.bsu.parsing;
import by.bsu.xmlstudents.Student;
import java.io.IOException;
import java.util.Set;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
public class StudentsSAXBuilder {
          private Set<Student> students;
          private StudentHandler sh;
          private XMLReader reader;
          public StudentsSAXBuilder() {
                    // создание SAX-анализатора
                    sh = new StudentHandler();
                    try {
                              // создание объекта-обработчика
                              reader = XMLReaderFactory.createXMLReader();
```

```
                    reader.setContentHandler(sh);
                } catch (SAXException e) {
                    System.err.print("ошибка SAX парсера: " + e);
                }
        }
        public Set<Student> getStudents() {
                return students;
        }
        public void buildSetStudents(String fileName) {
                try {
                    // разбор XML-документа
                    reader.parse(fileName);
                } catch (SAXException e) {
                    System.err.print("ошибка SAX парсера: " + e);
                } catch (IOException e) {
                    System.err.print("ошибка I/O потока: " + e);
                }
                students = sh.getStudents();
        }
}
package by.bsu.parsing;
public enum StudentEnum {
    STUDENTS("students"),
    LOGIN("login"),
    FACULTY("faculty"),
    STUDENT("student"),
    NAME("name"),
    TELEPHONE("telephone"),
    COUNTRY("country"),
    CITY("city"),
    STREET("street"),
    ADDRESS("address");
    private String value;
    private StudentEnum(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }
}
package by.bsu.xmlstudents;
import java.util.EnumSet;
import java.util.HashSet;
import java.util.Set;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
public class StudentHandler extends DefaultHandler {
        private Set<Student> students;
```

```
private Student current = null;
private StudentEnum currentEnum = null;
private EnumSet<StudentEnum> withText;
public StudentHandler() {
          students = new HashSet<Student>();
          withText = EnumSet.range(StudentEnum.NAME, StudentEnum.STREET);
}
public Set<Student> getStudents() {
          return students;
}
public void startElement(String uri, String localName, String qName, Attributes attrs) {
          if ("student".equals(localName)) {
                    current = new Student();
                    current.setLogin(attrs.getValue(0));
                    if (attrs.getLength() == 2) {
                              current.setFaculty(attrs.getValue(1));
                    }
          } else {
                   StudentEnum temp = StudentEnum.valueOf(localName.toUpperCase());
                    if (withText.contains(temp)) {
                              currentEnum = temp;
                    }
          }
}
public void endElement(String uri, String localName, String qName) {
          if ("student".equals(localName)) {
                    students.add(current);
          }
}
public void characters(char[] ch, int start, int length) {
          String s = new String(ch, start, length).trim();
          if (currentEnum != null) {
                switch (currentEnum) {
                    case NAME:
                              current.setName(s);
                              break;
                    case TELEPHONE:
                              current.setTelephone(Integer.parseInt(s));
                              break;
                    case STREET:
                              current.getAddress().setStreet(s);
                              break;
                    case CITY:
                              current.getAddress().setCity(s);
                              break;
                    case COUNTRY:
                              current.getAddress().setCountry(s);
                              break;
                    default:
```

```
                                    throw new EnumConstantNotPresentException(
                                    currentEnum.getDeclaringClass(), currentEnum.name());
                            }
                        }
                        currentEnum = null;
                }
        }
```

/ # 20 # создание и запуск парсера # */*

```
StudentsSAXBuilder saxBuilder = new StudentsSAXBuilder();
saxBuilder.buildSetStudents("data/students.xml");
System.out.println(saxBuilder.getStudents());
```

В результате на консоль будет выведено следующее множество описаний объектов:

[
Login: MitarAlex7
Name: Mitar Alex
Telephone: 2456474
Faculty: mmf
Address:
Country: Belarus
City: Minsk
Street: Kalinovsky 45
,
Login: Pashkin5
Name: Pashkin Alex
Telephone: 3453789
Faculty: mmf
Address:
Country: Belarus
City: Brest
Street: Knorina 56
]

Древовидная модель

Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода, однако при совместном использовании библиотек с аналогичными классами следует следить за совместимостью и корректностью взаимодействия.

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

В пакете **org.w3c.dom** можно найти интерфейсы, представляющие указанные объекты. 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: Xerces и JAXP, который входит в JDK.

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

DOM JAXP

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, замечательная особенность которого — 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. Этот 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;

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 элементом XML-документа и его содержимым. 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;

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;

boolean isId() — проверяет атрибут на тип ID.

Ниже приведен стандартный способ разбора документа **students.xml** с использованием DOM-анализатора и инициализация на его основе множества объектов.

/ # 21 # создание обьектов на основе экземпляра Document # StudentsDOMBuilder.java */*

```
package by.bsu.xmlstudents;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
```

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class StudentsDOMBuilder {
private Set<Student> students;
          private DocumentBuilder docBuilder;
          public StudentsDOMBuilder() {
               this.students = new HashSet<Student>();
               // создание DOM-анализатора
               DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
               try {
                              docBuilder = factory.newDocumentBuilder();
               } catch (ParserConfigurationException e) {
                              System.err.println("Ошибка конфигурации парсера: " + e);
               }
          }
          public Set<Student> getStudents() {
          return students;
          }
          public void buildSetStudents(String fileName) {
               Document doc = null;
               try {
                              // parsing XML-документа и создание древовидной структуры
                              doc = docBuilder.parse(fileName);
                              Element root = doc.getDocumentElement();
               // получение списка дочерних элементов <student>
               NodeList studentsList = root.getElementsByTagName("student");
                              for (int i = 0; i < studentsList.getLength(); i++) {
                                        Element studentElement = (Element) studentsList.item(i);
                                        Student student = buildStudent(studentElement);
                                        students.add(student);
                              }
               } catch (IOException e) {
                              System.err.println("File error or I/O error: " + e);
               } catch (SAXException e) {
                              System.err.println("Parsing failure: " + e);
               }
          }
          private Student buildStudent(Element studentElement) {
               Student student = new Student();
               // заполнение объекта student
               student.setFaculty(studentElement.getAttribute("faculty")); // проверка на null
               student.setName(getElementTextContent(studentElement, "name"));
               Integer tel = Integer.parseInt(getElementTextContent(
                                        studentElement,"telephone"));
                    student.setTelephone(tel);
                    Student.Address address = student.getAddress();
                    // заполнение объекта address
                    Element adressElement = (Element) studentElement.getElementsByTagName(
```

```
                                        "address").item(0);
            address.setCountry(getElementTextContent(adressElement, "country"));
            address.setCity(getElementTextContent(adressElement, "city"));
            address.setStreet(getElementTextContent(adressElement, "street"));
            student.setLogin(studentElement.getAttribute("login"));
            return student;
        }
        // получение текстового содержимого тега
        private static String getElementTextContent(Element element, String elementName) {
            NodeList nList = element.getElementsByTagName(elementName);
            Node node = nList.item(0);
            String text = node.getTextContent();
            return text;
        }
}
```

/ # 22 # создание и запуск парсера # */*

```
StudentsDOMBuilder domBuilder = new StudentsDOMBuilder();
domBuilder.buildSetStudents("data/students.xml");
System.out.println(domBuilder.getStudents());
```

Создание XML–документа

Документы можно не только читать, но также модифицировать и создавать совершенно новые. Для этого необходимо создать объекты классов **Document**, **Element**, добавить к последнему атрибуты и текстовое содержимое, после чего присоединить их к объекту, который в дереве XML-документа находится выше.

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

/ # 23 # создание и запись документа # CreateDocument.java */*

```
package by.bsu.transform;
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class CreateDocument {
```

```
        public static void main(String[] args) {
                DocumentBuilderFactory documentBuilderFactory =
                                                DocumentBuilderFactory.newInstance();
                DocumentBuilder documentBuilder = null;
                try {
                        documentBuilder =
                                documentBuilderFactory.newDocumentBuilder();
                } catch (ParserConfigurationException e) {
                        e.printStackTrace();
                }
                Document document = documentBuilder.newDocument();
                String root = "book";
                Element rootElement = document.createElement(root);
                document.appendChild(rootElement);
                for (int i = 0; i < 1; i++) {
                        String elementName = "name";
                        Element emName = document.createElement(elementName);
                        String name = "Technique Java";
                        emName.appendChild(document.createTextNode(name));

                        String elementAuthor = "author";
                        Element emAuthor = document.createElement(elementAuthor);
                        String author = "Blinov";
                        emAuthor.appendChild(document.createTextNode(author));
                        emAuthor.setAttribute("id", "3");
                        rootElement.appendChild(emName);
                        rootElement.appendChild(emAuthor);
                }
                TransformerFactory transformerFactory = TransformerFactory.newInstance();
                try {
                   Transformer transformer = transformerFactory.newTransformer();
                   DOMSource source = new DOMSource(document);
                   StreamResult result = new StreamResult(new FileWriter("data/book.xml"));
                   transformer.transform(source, result);
                } catch (TransformerConfigurationException e) {
                   e.printStackTrace();
                } catch (TransformerException e) {
                   e.printStackTrace();
                } catch (IOException e) {
                   e.printStackTrace();
                }
        }
}
```

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

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<book>
        <name>Technique Java</name>
        <author id="3">Blinov</author>
</book>
```

StAX

StAX (Streaming API for XML), который еще называют pull-парсером, включен в JSDK, начиная с версии Java 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 указывает 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 и расположены в пакете **javax.xml.stream**. Для чтения XML требуется получить ссылку на экземпляр **XMLStreamReader**:

```
StringReader input = new StringReader(fileName); // из пакета java.io
```

или

```
InputStream input = new FileInputStream(new File(fileName));
```

далее

```
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory.createXMLStreamReader(input);
```

после чего по экземпляру **XMLStreamReader** можно организовать навигацию аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **next()**:

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

int next() — переходит к следующей вершине-константе XML, извлекая тип текущей.

При попытке вызове на константе несоответствующего ей метода генерируется исключительная ситуация **IllegalStateException**.

Чаще всего данные извлекаются с применением методов:

String getLocalName() — возвращает название тега (элемента) для текущей константы;

String getAttributeValue(String namespaceURI, String localName) — возвращает значение атрибута по имени;

String getAttributeValue(ine index) — возвращает значение атрибута по номеру позиции;

String getText() — возвращает текст для констант **CHARACTERS, CDATA, COMMENT** и др.

Возможные типы вершин и методы, применимые к ним (см. таблицу ниже).

Типы вершин-констант	Методы
Для всех типов констант	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(), getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag() getAttributeXXX(), isAttributeSpecified(),
NAMESPACE	next(), nextTag() getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS	next(), getTextXXX(), nextTag()
CDATA	next(), getTextXXX(), nextTag()
COMMENT	next(), getTextXXX(), nextTag()
SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()

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

/ # 24 # реализация разбора XML-документа на основе StAX # StudentsStAXBuilder.java */*

```
package by.bsu.xmlstudents;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
public class StudentsStAXBuilder {
          private HashSet<Student> students = new HashSet<>();
          private XMLInputFactory inputFactory;
          public StudentsStAXBuilder() {
```

```
            inputFactory = XMLInputFactory.newInstance();
    }
    public Set<Student> getStudents() {
            return students;
    }
    public void buildSetStudents(String fileName) {
            FileInputStream inputStream = null;
            XMLStreamReader reader = null;
            String name;
            try {
               inputStream = new FileInputStream(new File(fileName));
               reader = inputFactory.createXMLStreamReader(inputStream);
               // StAX parsing
               while (reader.hasNext()) {
                 int type = reader.next();
                 if (type == XMLStreamConstants.START_ELEMENT) {
                      name = reader.getLocalName();
                   if (StudentEnum.valueOf(name.toUpperCase()) == StudentEnum.STUDENT) {
                       Student st = buildStudent(reader);
                       students.add(st);
                   }
                 }
               }
            } catch (XMLStreamException ex) {
                      System.err.println("StAX parsing error! " + ex.getMessage());
            } catch (FileNotFoundException ex) {
                      System.err.println("File " + fileName + " not found! " + ex);
            } finally {
                      try {
                                if (inputStream != null) {
                                          inputStream.close();
                                }
                      } catch (IOException e) {
                          System.err.println("Impossible close file "+fileName+" : "+e);
                      }
            }

    }
    private Student buildStudent(XMLStreamReader reader) throws XMLStreamException {
        Student st = new Student();
        st.setLogin(reader.getAttributeValue(null, StudentEnum.LOGIN.getValue()));
        st.setFaculty(reader.getAttributeValue(null,StudentEnum.FACULTY.getValue())); // проверить на null

        String name;
        while (reader.hasNext()) {
                  int type = reader.next();
                  switch (type) {
                     case XMLStreamConstants.START_ELEMENT:
                            name = reader.getLocalName();
                            switch (StudentEnum.valueOf(name.toUpperCase())) {
```

```
                                case NAME:
                                        st.setName(getXMLText(reader));
                                        break;
                                case TELEPHONE:
                                        name = getXMLText(reader);
                                        st.setTelephone(Integer.parseInt(name));
                                        break;
                                case ADDRESS:
                                        st.setAddress(getXMLAddress(reader));
                                        break;
                                }
                                break;
                        case XMLStreamConstants.END_ELEMENT:
                                name  =  reader.getLocalName();
                                if (StudentEnum.valueOf(name.toUpperCase()) == StudentEnum.STUDENT) {
                                        return st;
                                }
                                break;
                        }

                }
                throw new XMLStreamException("Unknown element in tag Student");
        }
        private Student.Address getXMLAddress(XMLStreamReader reader) throws XMLStreamException {

                Student.Address address = new Student.Address();
                int type;
                String name;
                while (reader.hasNext()) {
                        type = reader.next();
                        switch (type) {
                           case XMLStreamConstants.START_ELEMENT:
                                name = reader.getLocalName();
                                switch (StudentEnum.valueOf(name.toUpperCase())) {
                                   case COUNTRY:
                                        address.setCountry(getXMLText(reader));
                                        break;
                                   case CITY:
                                        address.setCity(getXMLText(reader));
                                        break;
                                   case STREET:
                                        address.setStreet(getXMLText(reader));
                                        break;
                                }
                                break;
                           case XMLStreamConstants.END_ELEMENT:
                                name  =  reader.getLocalName();
                                if (StudentEnum.valueOf(name.toUpperCase()) == StudentEnum.ADDRESS){
                                        return address;
```

```
                                        }
                                        break;
                                }
                        }
                        throw new XMLStreamException("Unknown element in tag Address");
                }
                private String getXMLText(XMLStreamReader reader) throws XMLStreamException {
                        String text = null;
                        if (reader.hasNext()) {
                                reader.next();
                                text = reader.getText();
                        }
                        return text;
                }
}
```

Для запуска приложения разбора документа с помощью StAX ниже приведен достаточно простой код, аналогичный коду запуска SAX и DOM парсеров.

/ # 25 # создание и запуск StAX-парсера # */*

```
StudentsStAXBuilder staxBuilder = new StudentsStAXBuilder();
staxBuilder.buildSetStudents("data/students.xml");
System.out.println(staxBuilder.getStudents());
```

Обработку документов с помощью всех трех парсеров можно объединить в одно приложение с использованием шаблонов проектирования **Factory Method** и **Builder**. Для этого будет построена иерархия классов-builder-ов во главе с абстрактным классом **AbstractStudentsBuilder** в виде:

/ # 26 # вершина иерархии builder-ов # AbstractStudentsBuilder.java */*

```
package by.bsu.xmlstudents;
import java.util.HashSet;
import java.util.Set;
public abstract class AbstractStudentsBuilder {
          // protected так как к нему часто обращаются из подкласса
          protected Set<Student> students;
          public AbstractStudentsBuilder() {
                    students = new HashSet<Student>();
          }
          public AbstractStudentsBuilder(Set<Student> students) {
                    this.students = students;
          }
          public Set<Student> getStudents() {
                    return students;
          }
          abstract public void buildSetStudents(String fileName);
}
```

Приведенные выше классы разбора XML-документов с помощью SAX, DOM, StAX следует адаптировать вследствие определения для них абстрактного класса и передачи ему атрибута **Set<Student> students** и метода **getStudents()**.

/ # 27 # реализации конкретных bulder-ов # StudentsSAXBuilder.java # StudentsDOMBuilder.java # StudentsStAXBuilder.java */*

```
public class StudentsSAXBuilder extends AbstractStudentsBuilder {
          private StudentHandler sh;
          private XMLReader reader;
          public StudentsSAXBuilder() {
                    // more code
          }
          public StudentsSAXBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
public class StudentsDOMBuilder extends AbstractStudentsBuilder {
          private DocumentBuilder docBuilder;
          public StudentsDOMBuilder() {
                    // more code
          }
          public StudentsDOMBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
public class StudentsStAXBuilder extends AbstractStudentsBuilder {
          private XMLInputFactory inputFactory;
          public StudentsStAXBuilder() {
                    // more code
          }
          public StudentsStAXBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
```

Шаблон **Factory Method** предназначен для создания объектов, находящихся в иерархической зависимости. В данной ситуации класс, реализующий шаблон, будет производить экземпляры подклассов абстрактного класса **AbstractStudentsBuilder**. Принятие решения о конкретном типе будет производиться на основании передаваемого в factory-метод строкового значения с именем желаемого парсера.

/ # 28 # фабрика для создания конкретных bulder-ов # StudentBuilderFactory.java */*

```
public class StudentBuilderFactory {
          private enum TypeParser {
                    SAX, STAX, DOM
          }
          public AbstractStudentsBuilder createStudentBuilder(String typeParser) {
                    TypeParser type = TypeParser.valueOf(typeParser.toUpperCase());
                    switch (type) {
                        case DOM:
                              return new StudentsDOMBuilder();
                        case STAX:
                              return new StudentsStAXBuilder();
                        case SAX:
                              return new StudentsSAXBuilder();
                        default:
                              throw new EnumConstantNotPresentException (type.getDeclaringClass(), type.name());

                    }
          }
}
```

Запускать приложение из метода **main()** стало возможным с помощью кода:

/ # 29 # запуск приложения с выбором парсеров */*

```
StudentBuilderFactory sFactory = new StudentBuilderFactory();
AbstractStudentsBuilder builder = sFactory.createStudentBuilder("stax");
builder.buildSetStudents("data/students.xml");
System.out.println(builder.getStudents());
```

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 собой 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 текстовый документ другого типа.

Пусть требуется построить 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 проблемы. В следующем коде приведен файл таблицы стилей **students.xsl**, решающий поставленную задачу.

 # 30 # документ-стиль для преобразования # students.xsl

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

 # 31 # стиль с ATV # students.xsl

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

/ # 32 # трансформация XML # SimpleTransform.java */*

```
package by.bsu.transform;
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("data/students.xsl"));
         // установка исходного XML-документа и конечного XML-файла
         transformer.transform(new StreamSource("data/students.xml"),
                                                    new StreamResult("newstudents.xml"));
         System.out.println("Transform " + fileName + " complete");
         } catch(TransformerException e) {
               System.err.println("Impossible transform file " + fileName + " : " + ex);
         }
     }
}
```

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>Pashkin 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 способ (правила) трансформации одного документа в другой.

Какие же элементы используются в данном листинге?

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

Инструкция говорит о том, что конечный документ, который получится после преобразования, будет 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 **student**, содержащему атрибут **login** со значением **base:**

```
<xsl:template match="student[@login='base']">
```

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 узлов-листьев. На каждом шаге процессор пытается применить к данному элементу некоторый шаблон преобразования; если в таблице стилей для текущего просматриваемого элемента есть нужный 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 14

1. Создать файл XML и соответствующую ему схему XSD.
2. При разработке XSD использовать простые и комплексные типы, перечисления, шаблоны и предельные значения.
3. Сгенерировать класс, соответствующий данному описанию.
4. Создать приложение для разбора XML-документа и инициализации коллекции объектов информацией из XML-файла. Для разбора использовать SAX, DOM и StAX парсеры. Для сортировки объектов использовать интерфейс Comparator.
5. Произвести проверку XML-документа с привлечением XSD.
6. Определить метод, производящий преобразование разработанного XML-документа в документ, указанный в каждом 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) — предпочтительные условия произрастания: температура (в градусах), освещение (светолюбиво либо нет), полив (мл в неделю);
   — Multiplying — размножение: листьями, черенками либо семенами.
   Корневой элемент назвать Flower.
   С помощью XSL преобразовать XML-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.
   С помощью XSL преобразовать XML-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, поминутная), плата за подключение к тарифу (0–n рублей).
   Корневой элемент назвать Tariff.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать тарифы по абонентской плате.
4. **Лекарственные препараты**.
   Лекарственные препараты имеют следующие характеристики:
   — Name — наименование препарата;
   — Pharm — фирма-производитель;
   — Group — группа препаратов, к которым относится лекарство (антибиотики, болеутоляющие, витамины и т. п.);
   — Analogs (может быть несколько) — содержит наименование аналога;
   — Versions — варианты исполнения (консистенция/вид: таблетки, капсулы, порошок, капли и т. п.). Для каждого варианта исполнения может быть несколько производителей лекарственных препаратов со следующими характеристиками:
     • Certificate — свидетельство о регистрации препарата (номер, даты выдачи/истечения действия, регистрирующая организация);
     • Package — упаковка (тип упаковки, количество в упаковке, цена за упаковку);
     • Dosage — дозировка препарата, периодичность приема;
   Корневой элемент назвать Medicine.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать лекарства по цене.
5. **Компьютеры**.
   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.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать Critical.

6. **Электроинструменты**.
Электроинструменты можно структурировать по следующей схеме:
— Model — название модели;
— Handy — одно- или двуручное;
— Origin — страна производства;
— TC (должно быть несколько) — технические характеристики: энергопотребление (низкое, среднее, высокое), производительность (в единицах в час), возможность автономного функционирования и т. д.;
— Material — материал изготовления.
Корневой элемент назвать PowerTools или Power.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать страну производства.

7. **Столовые приборы**.
Столовые приборы можно структурировать по следующей схеме:
— Type — тип (нож, вилка, ложка и т. д.);
— Origin — страна производства;
— Visual (должно быть несколько) — визуальные характеристики: лезвие, зубец (длина лезвия, зубца [10– n см], ширина лезвия [10–n мм]), материал (лезвие [сталь, чугун, медь и т. д.]), рукоять (деревянная [если да, то указать тип дерева], пластик, металл);
— Value — коллекционный либо нет.
Корневой элемент назвать FlatWare.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по длине лезвия, зубца, объему.

8. **Самолеты**.
Самолеты можно описать по следующей схеме:
— Model — название модели;
— Origin — страна производства;
— Chars (должно быть несколько) — характеристики, могут быть следующими: тип (пассажирский, грузовой, почтовый, пожарный, сельскохозяйственный), количество мест для экипажа, характеристики (грузоподъемность, число пассажиров), наличие радара;
— Parameters — длина (в метрах), ширина (в метрах), высота (в метрах);
— Price — цена (в талерах).
Корневой элемент назвать Plane.

С помощью XSL преобразовать XML-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 г), жиры (в г) и углеводы (в г);
   — Production — предприятие-изготовитель.
   Корневой элемент назвать Candy.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по месту изготовления.
10. **Пиво**.
   — Name — название;
   — Type — тип пива (темное, светлое, лагер, живое);
   — Al — алкогольное либо безалкогольное;
   — Manufacturer — фирма-производитель;
   — Ingredients (должно быть несколько) — ингредиенты: вода, солод, хмель, сахар и т. д.;
   — Chars (должно быть несколько) — характеристики: количество оборотов (если алкогольное), прозрачность (в процентах), фильтрованное либо нет, пищевая ценность (ккал), способ разлива (объем и материал емкостей).
   Корневой элемент назвать Beer.
   С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать производителя.
11. **Периодические издания**.
   — Title — название;
   — Type — тип (газета, журнал, буклет);
   — Monthly — переодичность выхода;
   — Chars (должно быть несколько) — характеристики: цветность (да либо нет), объем (n страниц), глянцевое (да [только для журналов и буклетов] либо нет [для газет]), подписной индекс (только для газет и журналов).
   Корневой элемент назвать Paper.
   С помощью XSL преобразовать XML-файл в формат plain text, при выводе организовать подачу информации в удобном для прочтения виде.
12. **Туристические путевки.**
   Туристические путевки, предлагаемые агентством, имеют следующие характеристики:
   — Type — тип (выходного дня, экскурсионная, отдых, паломничество и т. д.);
   — Country — страна, выбранная для путешествия;
   — Number days/nights — количество дней и ночей;

— Transport — вид перевозки туристов (авиа, ж/д, авто, лайнер);
— Hotel characteristic (должно быть несколько) — количество звезд, включено ли питание и какое (HB, BB, Al), какой номер (1-, 2-, 3-местные), есть ли телевизор, кондиционер и т. д.;
— Cost — стоимость путевки (сколько и что включено).

Корневой элемент назвать Tourist voucher.

С помощью XSL преобразовать XML-файл в формат HTML, с выводом информации в табличном виде.

13. **Старые открытки**.
— Thema — тема изображения (городской пейзаж, природа, люди, религия, спорт, архитектура...);
— Type — тип (поздравительная, рекламная, обычная). Была ли отправлена;
— Country — страна производства;
— Year — год издания;
— Author — имя автора/ов (если известен);
— Valuable — историческая, коллекционная или тематическая ценность.

Корневой элемент назвать Old Card.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в отдельную страницу для каждого концерта.

14. **Банковские вклады**.
— Name — название банка;
— Country — страна регистрации;
— Type — тип вклада (до востребования, срочный, расчетный, накопительный, сберегательный, металлический);
— Depositor — имя вкладчика;
— Account id — номер счета;
— Amount on deposit — сумма вклада;
— Profitability — годовой процент;
— Time constraints — срок вклада.

Корневой элемент назвать Bank.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в табличном виде.