In XML ToDo lists:

1. Rearrange the document with details
2. External DTD validation process
3. XML Schema validation example
4. Sax parser example
5. Dom Parser example
6. XML stand for eXtensible Markup Language.

* XML is a software and hardware-independent tool for carrying information.
* XML is text-based markup language derived from Standard Generalized Markup Language (SGML).

XML is designed to store and transport data i.e. (describe the data)

* With XML, each individual piece of information is "marked up" (a marker shows the meaning of the associated data) with a tag that attaches meaning to the information.
* XML handles data in a tree structure having one-and only one-root element

1. XML can has sender and receiver information, it also has a heading and a message body.

But still, the XML document does not DO anything. It is just information wrapped in tags. Someone must write a piece of software to send, receive or display it.

1. SGML(Standard Generalized Markup Language)

1. Difference between XML and HTML:

|  |  |
| --- | --- |
| **HTML** | **XML** |
| 1. Html is used for presenting data in the required format. | 1. XML is mainly used for storing data. |
| 1. HTML document may not be well formed document. | 2. Xml document is well formed document. |
| 1. HTML has set of predefined tags, whose meanings can’t be changed. | 3. Xml contains no predefined tags. |
| 1. We can’t define our tag using HTML. | 4. We must define our own tag as per the requirement. |
| 1. It’s not case sensitive. | 5. it’s case sensitive. |

Question: What is well formed document?

Answer:

1. XML documents must have a root element i.e. single root element.
2. XML document should form a tree structure.
3. XML elements must have a closing tag for each start tag.
4. XML tags are case sensitive.
5. XML elements must be properly nested.
6. XML attribute values must be quoted i.e. inside “” or ‘’.

An XML document should start with a header such as

<?xml version="1.0"?>

or

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

Strictly speaking, a header is optional, but it is highly recommended.

Structure of XML Document:

XML document contains three parts:

1. Prolog
2. Body
3. Epilog
4. Prolog:
5. Prolog is the first section of XML document and it’s mandatory.
6. Prolog has three parts:
7. XML Declaration
8. Procession Instructions
9. Comments
10. XML Declaration:

<? xml version=”1.0” encoding = “…….” standalone = “…….” ?>

The ﬁrst line in an XML document is called the **prolog**:

* Version attribute is mandatory.
* Encoding and standalone attributes are optional.

encoding: Any valid character set.

standalone: yes/no

* Standalone attribute is to represent whether DTD declaration provided inside the XML or outside the XML separately.
* Default value of standalone attribute is “NO” which means that DTD rules are defined outside the XML document in a separate file.
* If we specify “YES” then we must specify the DTD rules within the XML document.

1. Procession instruction:

There are various procession instruction which can write after the xml declaration section.

1. Including DTDs
2. Including Schemas
3. Including XSL files
4. Comments:

Same as HTML comments:

<!-- Some comment -->

1. Body :

Body element start with root element and mainly for storing the data.

the body of the XML document contains the *root element,* which can contain other

elements.

1. Epilog:

Epilog is the last section of XML document and is completely optional.

It contains two parts:

1. Processing Instructions:
2. Comments

Example:

**employee.xml**

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

<employees>

<employee>

<id>1001</id>

<name>abc</name>

<salary>230000</salary>

<projectId>P001</projectId>

<dob>12-MAR-1976</dob>

<doj>11-APR-2001</doj>

</employee>

<employee>

<id>1002</id>

<name>xyz</name>

<salary>330000</salary>

<projectId>P002</projectId>

<dob>12-JAN-1977</dob>

<doj>09-JUL-2002</doj>

</employee>

<employee>

<id>1003</id>

<name>mnop</name>

<salary>300000</salary>

<projectId>P003</projectId>

<dob>12-MAR-1975</dob>

<doj>10-JUN-2000</doj>

</employee>

</employees>

Validating XML Document:

Following are rules to validate the xml document.

1. Xml

..

..

* You can validate the XML document using the following:

1. DTD ( Document Type Definition)
2. Schema (XML-based alternative to DTD)
3. DTD

When you define the DTD for your XML documents you need to use following:

1. <!ELEMENT>
2. <!ATTLIST>
3. <!ENTITY>
4. Cardinality
5. #PCDATA
6. CDATA
7. <!ELEMENT>

This tag is used to define elements of your xml document.

For Example:

<!ELEMENT employee (id,name,salary,projectId,dob,doj)>

<!ELEMENT id (#PCDATA)>

1. <!ATTLIST>

This is to specify the attribute information of one particular element.

For Example:

<!ATTLIST student

sid CDATA #REQURIED

job (yes/no) #IMPLIED

fee CDATA #IMPLIED >

IMPLIED means optional.

<!ATTLIST element-name attribute-name attribute-type attribute-value>

<!ATTLIST element-name attribute-name attribute-type #REQUIRED>

<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

<!ATTLIST element-name attribute-name attribute-type #FIXED "value">

<!ATTLIST element-name attribute-name (en1|en2|..) default-value>

DTD:  
<!ATTLIST payment type (check|cash) "cash">  
  
XML example:  
<payment type="check" />  
or  
<payment type="cash" />

DTD example:  
  
<!ATTLIST payment type CDATA "check">  
  
XML example:  
  
<payment type="check" />

The attribute-value can be one of the following:

|  |  |
| --- | --- |
| Value | Explanation |
| Value | The default value of the attribute |
| #REQUIRED | The attribute is required |
| #IMPLIED | The attribute is optional |
| #FIXED value | The attribute value is fixed |

1. <!ENTITY>

This is to define entity references.

For Example:

<!ENTITY >

…………..

1. Cardinality:

Represents occurrence of elements.

\*=> 0 or more

+=>1 or more

?=> 0 or 1

No symbol => 0 or 1

Symbol Meaning Example

, AND header (sender, recipient\*, date)

| OR message (email | letter)

() Occurs only Once (email | letter)

+ must occur at least once (header, subject?, text+)

? occurs either once or not at all (header, recipient\* , date?)

\* can occur zero or more times (sender, recipient\*, date)

For Example:

<!ELEMENT student (sname, email+, company?, phone\*)>

<!ELEMENT root (student\*, course+)>

* There are two type of data type in DTD.

1. PCDATA : used for elements.

* With PCDATA entity references will be resolved.

1. CDATA: used for attributes.

* With CDATA entity references will not be resolved.

Question: How to specify the DTD in an XML document?

Answer:

There are two ways to specify DTD in an XML document.

1. Internal DTDs
2. External DTDs
3. Internal DTDs:

Syntax:

<!DOCTYPE rootElement [

DTD Specification ]>

* You can specify the document type definitions inside the xml document itself.
* For specifying DTDs you must specify the standalone value as “yes”. (default value is “no”).
* This not recommendable because you can’t reuse the DTDs:

For Example:

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

<!DOCTYPE books[

<!-- <!element > can't use in lowercase -->

<!ELEMENT books (book+)>

<!ELEMENT book (bname, author)>

<!ELEMENT bname ( #PCDATA)>

<!ELEMENT author (#PCDATA)>

<!-- <!attlist> -->

<!ATTLIST book bid CDATA #IMPLIED>

]>

<books>

<book>

<bname>abc</bname>

<author>xyz</author>

</book>

<book bid=*"B-101"*>

<bname>pqrs</bname>

<author>mnop</author>

</book>

</books>

* <!ELEMENT books (book+)>

Between “books” and “(” space is required, otherwise it won’t be validated successfully.

* #PCDATA

Between “#” and “PCDATA” space shouldn’t be there.

* We can view the validation of the xml file by DTD using Internal Explorer 5.0 or higher.

If the validation of the xml file against the DTD is correct then only we can view the xml file using the IE.

* There are also online resources, such as this online XML Validator, but you need an internal DTD or a DTD available on the Internet in order to compare it against.

Now that you have the XML linked, you'll need a full parser to validate the XML files. While most browsers can check for well-formed XML, only Internet Explorer 5.0 and higher has a complete XML parser built in to the program that checks against DTDs. You can also use programs such as Dreamweaver, Cooktop, and a variety of other XML authoring software.

1. External DTDs:
2. You can define the DTD separately in separate file with the extension “.dtd”.
3. Externally defined DTD can be included in the XML document using doctype declaration.
4. If any elements, attributes, or entities are used in the XML document that are referenced or defined in an external DTD, standalone="no" must be included in the [XML declaration](https://xmlwriter.net/xml_guide/xml_declaration.shtml)

**Private" External DTDs:**

Private external DTDs are identified by the keyword SYSTEM, and are intended for use by a single author or group of authors.

<!DOCTYPE root\_element SYSTEM "DTD\_location">

where:

DTD\_location: relative or absolute URL

book.dtd

<>

- “.dtd” document is not an xml document.

Syntax for external doc type declaration:

<!DOCTYPE rootElement SYSTEM file>

<!DOCTYPE rootElement PUBLIC Identifier file>

* Including DTD inside XML

1. “.dtd” file in current folder
2. “.dtd” in different folder
3. “.dtd” is on www.
4. “.dtd” file in current folder

<!DOCTYPE books SYSTEM “book.dtd”>

<!DOCTYPE books PUBLIC “book” “book.dtd”>

1. “.dtd” in different folder

<!DOCTYPE books SYSTEM “file://D:/programs/book.dtd”>

<!DOCTYPE books PUBLIC “book” “file://D:/programs/book.dtd”>

1. “.dtd” is on www.

<!DOCTYPE books SYSTEM “http://www.book.com/book.dtd”>

<!DOCTYPE books PUBLIC “book” “http://www.book.com/book.dtd”>

* If location is not found identifier provides the alternative location in the current file system.

**XML Namespaces:**

* XML namespace is a name which will be used to resolve naming conflict.
* It is bit similar to java package.

student.xml

<students>

<student>

<name>xyz</name>

</student>

<course>

<name>abc</name>

</course>

</students>

<?xml version=”1.0”?>

<detail xmlns = “http://www.detail.com”

xmlns:stu = “http://www.detail.com/student”

xmlns:cou = “http://www.detail.com/course”>

<stu:student>

<stu:name>abc</stu:name>

</stu:student>

<cou:course>

<cou:name>xyz</cou:name>

</cou:course>

</detail>

XML Schema:

1. Xml Schema is an alternative to DTD for validating xml document.
2. XML Schema is advanced than dtds and has more features.
3. XML schema supports various datatypes like integer, string, data, float, boolean, etc
4. Schema allows to define custom complex data types.
5. Schema supports xml namespace.
6. Schema document is an xml document and must be well formed.
7. Schema definition should be placed in separate file with extension “.xsd”(xml schema definition)

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema:

* defines elements that can appear in a document
* defines attributes that can appear in a document
* defines which elements are child elements
* defines the order of child elements
* defines the number of child elements
* defines whether an element is empty or can include text
* defines data types for elements and attributes
* defines default and fixed values for elements and attributes

The XML Schema language is also referred to as XML Schema Definition (XSD).

A simple Xml document:

<?xml version="1.0"?>  
<note>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>

The following example is a DTD file called "note.dtd" that defines the elements of the XML document above ("note.xml"):

<!ELEMENT note (to, from, heading, body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>

The first line defines the note element to have four child elements: "to, from, heading, body".

Line 2-5 defines the to, from, heading, body elements to be of type "#PCDATA".

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

<?xml version="1.0"?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
targetNamespace="http://www.w3schools.com"  
xmlns="http://www.w3schools.com"  
elementFormDefault="qualified">  
  
<xs:element name="note">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="to" type="xs:string"/>  
      <xs:element name="from" type="xs:string"/>  
      <xs:element name="heading" type="xs:string"/>  
      <xs:element name="body" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>  
  
</xs:schema>

The note element is a **complex type** because it contains other elements. The other elements (to, from, heading, body) are **simple types** because they do not contain other elements.

Schema Element:

The <schema> element is the root element of every XML Schema:

<?xml version="1.0"?>  
  
<xs:schema>  
...  
...  
</xs:schema>

The <schema> element may contain some attributes. A schema declaration often looks something like this:

<?xml version="1.0"?>  
  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
targetNamespace="http://www.w3schools.com"  
xmlns="http://www.w3schools.com"  
elementFormDefault="qualified">  
...  
...  
</xs:schema>

xmlns:xs=<http://www.w3.org/2001/XMLSchema>

indicates that the elements and data types used in the schema come from the "http://www.w3.org/2001/XMLSchema" namespace. It also specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with **xs:**

targetNamespace=<http://www.w3schools.com>

indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.w3schools.com" namespace.

xmlns=<http://www.w3schools.com>

indicates that the default namespace is "http://www.w3schools.com".

elementFormDefault="qualified"

indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

**Difference between DTD and Schema**

1. XML Schema is namespace aware, while DTD is not.

2. XML Schemas are written in XML, while DTDs are not.

3. XML Schema is strongly typed, while DTD is not.

4. XML Schema has a wealth of derived and built-in data types that are not available in DTD.

5. XML Schema does not allow inline definitions, while DTD does.

**XML Parser:**

SAX stands for **Simple API for XML** and DOM stands for **Document Object Model**.  
Both of these are APIs to parse XML documents. There is no direct competition between DOM and SAX  
as both of them have their own strengths and weaknesses.

The following are some fundamental difference between a SAX and DOM model of parsing.

**SAX:**

1. Parses the document on node by node basis.  
2. Does not store the XML in memory.  
3. We cannot insert or delete a node.  
4. This model uses top to bottom traversing.  
5. This model does not preserve comments.  
6. It runs little faster than DOM

**DOM**

1. Stores the entire XML document into memory before processing  
2. Occupies more memory  
3. We can insert or delete nodes  
4. This model can traverse in any direction.  
5. This model preserves comments.  
6. It runs slower than SAX model

So, when to choose what model to use?

Here is what you can do.

If you just need to read a node, but do not require to insert/delte node, then use SAX.  
If you require node manipulation (insert/delete) nodes, use DOM.

There are various parsers in Java that can be used. The following is a list of few java parsers.

* Apache Parser [Xerces]
* Oracle XML Parser
* Sun XML Parser

All of these parsers provide SAX and DOM parsing mechanisms.

1. SAX parser (Simple API for XML)
2. DOM Parser
3. SAX Parser

SAX parser use callback function (org.xml.sax.helpers.DefaultHandler) to inform clients of the XML document structure.

SAX Parser is faster and uses less memory than DOM parser.

SAX callback methods :

* **startDocument()** and **endDocument()** – Method called at the start and end of an XML document.
* **startElement()** and **endElement()** – Method called at the start and end of a document element.

|  |  |
| --- | --- |
| **public** **void** startElement (String uri, String localName, String qName, Attributes attributes) **throws** SAXException | Receive notification of the start of an element.  By default, do nothing. Application writers may override this method in a subclass to take specific actions at the start of each element (such as allocating a new tree node or writing output to a file).  **Specified by:** [startElement(...)](eclipse-javadoc:%E2%98%82=Test/C:%5C/apps%5C/Java%5C/jdk1.6.0_37%5C/jre%5C/lib%5C/rt.jar%3Corg.xml.sax(ContentHandler.class%E2%98%83ContentHandler~startElement~Ljava.lang.String;~Ljava.lang.String;~Ljava.lang.String;~Lorg.xml.sax.Attributes;) in [ContentHandler](eclipse-javadoc:%E2%98%82=Test/C:%5C/apps%5C/Java%5C/jdk1.6.0_37%5C/jre%5C/lib%5C/rt.jar%3Corg.xml.sax(ContentHandler.class%E2%98%83ContentHandler)  **Parameters:**  **uri** The Namespace URI, or the empty string if the element has no Namespace URI or if Namespace processing is not being performed.  **localName** The local name (without prefix), or the empty string if Namespace processing is not being performed.  **qName** The qualified name (with prefix), or the empty string if qualified names are not available.  **attributes** The attributes attached to the element. If there are no attributes, it shall be an empty Attributes object. |
| **public** **void** endElement (String uri, String localName, String qName) **throws** SAXException | Receive notification of the end of an element.  By default, do nothing. Application writers may override this method in a subclass to take specific actions at the end of each element (such as finalising a tree node or writing output to a file).  **Parameters:**  **uri** The Namespace URI, or the empty string if the element has no Namespace URI or if Namespace processing is not being performed.  **localName** The local name (without prefix), or the empty string if Namespace processing is not being performed.  **qName** The qualified name (with prefix), or the empty string if qualified names are not available. |
| **public** **void** characters (**char** ch[], **int** start, **int** length)**throws** SAXException | Receive notification of character data inside an element.  By default, do nothing. Application writers may override this method to take specific actions for each chunk of character data (such as adding the data to a node or buffer, or printing it to a file).  **Parameters:**  **ch** The characters.  **start** The start position in the character array.  **length** The number of characters to use from the character array. |
| **public** **void** startDocument ()**throws** SAXException | Receive notification of the beginning of the document.  By default, do nothing. Application writers may override this method in a subclass to take specific actions at the beginning of a document (such as allocating the root node of a tree or creating an output file). |
| **public** **void** endDocument ()t**hrows** SAXException | Receive notification of the end of the document.  By default, do nothing. Application writers may override this method in a subclass to take specific actions at the end of a document (such as finalising a tree or closing an output file). |
|  |  |
|  |  |

**startElement() :** Every time a SAX parser gets a opening tag '<', it calls startElement() and process the xml according to the code written in startElement().  
  
**endElement():** Every time a SAX parser gets a closing tag '>', it calls endElement() and process the xml according to the code written in endElement().  
  
**character():** Every time a SAX parser gets a simple character string, it calls character() method and the xml according to the code written in startElement().

Steps to write the SAX parser:

1. prepare Xml file to be parsed
2. Create a model class.
3. Build Handler by extending DefaultHandler
4. Write the parser code.
5. Write the client code.

**public** **class** DefaultHandler **implements** EntityResolver, DTDHandler, ContentHandler, ErrorHandler

* Create a handler class by extending the “**org.xml.sax.helpers.DefaultHandler**” class and override below methods.

**public** **void** startElement (String uri, String localName, String qName, Attributes attributes) **throws** SAXException{}

**public** **void** endElement (String uri, String localName, String qName) **throws** SAXException{}

**public** **void** characters (**char** ch[], **int** start, **int** length)**throws** SAXException{}

**public** **void** startDocument ()**throws** SAXException{}

**public** **void** endDocument ()t**hrows** SAXException{}

1. prepare Xml file to be parsed

**customer1.xml**

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

<!DOCTYPE customers [

<!ELEMENT customers (customer\*)>

<!ELEMENT customer (id,name,mobile,city)>

<!ELEMENT id (#PCDATA)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT mobile (#PCDATA)>

<!ELEMENT city (#PCDATA)>

]>

<customers>

<customer>

<id>10001</id>

<name>dfsdfad</name>

<mobile>9898989898</mobile>

<city>Bangalore</city>

</customer>

<customer>

<id>10002</id>

<name>fggdfg</name>

<mobile>9898984444</mobile>

<city>Jaipur</city>

</customer>

<customer>

<id>10003</id>

<name>fgfgd</name>

<mobile>5678989898</mobile>

<city>Maysor</city>

</customer>

<customer>

<id>10004</id>

<name>dffdgfh</name>

<mobile>5454545454</mobile>

<city>Pune</city>

</customer>

<customer>

<id>10005</id>

<name>adfsdd</name>

<mobile>9896669898</mobile>

<city>Patna</city>

</customer>

</customers>

1. Create a model class.

**Customer.java**

**package** com.test.xmlSaxParser;

**public** **class** Customer {

**private** Integer id;

**private** String name;

**private** Long mobile;

**private** String city;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

**public** Long getMobile() {

**return** mobile;

}

**public** **void** setMobile(Long mobile) {

**this**.mobile = mobile;

}

@Override

**public** String toString() {

**return** "Customer [id=" + id + ", name=" + name + ", mobile=" + mobile

+ ", city=" + city + "]";

}

}

1. Build Handler by extending DefaultHandler

**CustomerHandler.java**

**package** com.test.xmlSaxParser;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.xml.sax.Attributes;

**import** org.xml.sax.SAXException;

**import** org.xml.sax.helpers.DefaultHandler;

**public** **class** CustomerHandler **extends** DefaultHandler {

**private** Customer c;

**private** **boolean** id;

**private** **boolean** name;

**private** **boolean** mobile;

**private** **boolean** city;

**private** ArrayList<Customer> customerList= **new** ArrayList<Customer>();

**public** **void** startElement(String uri, String localName, String qName, Attributes attributes) **throws** SAXException {

**if**("customer".equalsIgnoreCase(qName)){

c = **new** Customer();

} **else** **if**("id".equalsIgnoreCase(qName)){

id = **true**;

} **else** **if**("name".equalsIgnoreCase(qName)){

name = **true**;

} **else** **if**("mobile".equalsIgnoreCase(qName)){

mobile = **true**;

} **else** **if**("city".equalsIgnoreCase(qName)){

city = **true**;

}

}

**public** **void** endElement(String uri, String localName, String qName) **throws** SAXException {

**if**("customer".equalsIgnoreCase(qName)){

customerList.add(c);

}

}

**public** **void** characters(**char** ch[], **int** start, **int** length) **throws** SAXException {

String value = **new** String(ch, start, length);

**if**(id){

c.setId(Integer.*valueOf*(value));

id = **false**;

} **else** **if**(name){

c.setName(value);

name = **false**;

} **else** **if**(mobile){

c.setMobile(Long.*valueOf*(value));

mobile = **false**;

} **else** **if**(city){

c.setCity(value);

city = **false**;

}

}

**public** **void** startDocument() **throws** SAXException {

System.*out*.println("document started!");

}

**public** **void** endDocument() **throws** SAXException {

System.*out*.println("document ended!");

}

**public** List<Customer> getAllCustomer(){

**return** customerList;

}

}

1. Write the parser code.

**CustomerSAXReader**

**package** com.test.xmlSaxParser;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** org.xml.sax.InputSource;

**import** org.xml.sax.SAXException;

**import** org.xml.sax.XMLReader;

**import** org.xml.sax.helpers.XMLReaderFactory;

**public** **class** CustomerSaxReader {

**public** **static** **void** main(String[] args) {

**try** {

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

XMLReader reader = XMLReaderFactory.*createXMLReader*();

CustomerHandler ch = **new** CustomerHandler();

reader.setContentHandler(ch);

InputSource source = **new** InputSource(**new** FileInputStream(**new** File("src/customer1.xml")));

reader.parse(source);

System.*out*.println(ch.getAllCustomer());

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}**catch** (SAXException e) {

e.printStackTrace();

}

}

}

Output:

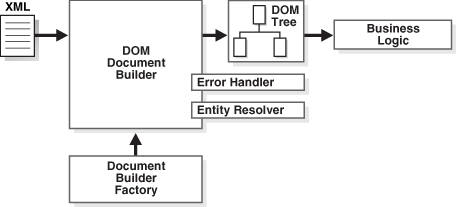
document started!

document ended!

[Customer [id=10001, name=dfsdfad, mobile=9898989898, city=Bangalore], Customer [id=10002, name=fggdfg, mobile=9898984444, city=Jaipur], Customer [id=10003, name=fgfgd, mobile=5678989898, city=Maysor], Customer [id=10004, name=dffdgfh, mobile=5454545454, city=Pune], Customer [id=10005, name=adfsdd, mobile=9896669898, city=Patna]]

1. DOM Parser:

DOM so called (Domain object model) is the most commonly used and easiest implementation to parse an xml in java. DOM loads the xml tree into memory before parsing it, this is the reason we need to have a large heap size to eliminate memory exceptions. In case we have a large xml to parse it is better to use SAX instead of DOM, because loading a large xml in memory is not a good choice.  
  
DOM is already a part of JDK, hence we don’t need to add any external jar to start with DOM.



Steps to write the SAX parser:

1. Import XML-related packages

You will need to import below packages first in your application.

|  |  |
| --- | --- |
| import org.w3c.dom.\*;  importjavax.xml.parsers.\*;  import java.io.\*; |  |

1. Create a DocumentBuilder

Next step is to create the DocumentBuilder object.

|  |  |
| --- | --- |
|  | DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  DocumentBuilder builder = factory.newDocumentBuilder(); |

1. Create a Document from a file

|  |  |
| --- | --- |
|  | Document document = builder.parse(new File( file )); |

1. Validate Document structure

This is optional but good to have it before start parsing.

|  |  |
| --- | --- |
|  | Schema schema = null;  try {    String language = XMLConstants.W3C\_XML\_SCHEMA\_NS\_URI;    SchemaFactory factory = SchemaFactory.newInstance(language);    schema = factory.newSchema(new File(name));  } catch (Exception e) {      e.printStackStrace();  }  Validator validator = schema.newValidator();  validator.validate(new DOMSource(document)); |

1. Extract the root element

You can get the root element from XML document using below code.

|  |  |
| --- | --- |
|  | Element root = document.getDocumentElement(); |

1. Examine attributes

You can examine the node attributes using below methods.

|  |  |
| --- | --- |
|  | element.getAttribute("attributeName") ;    //returns specific attribute  element.getAttributes();                //returns a Map (table) of names/values |

1. Examine sub-elements

Child elements can inquired in below manner.

|  |  |
| --- | --- |
|  | node.getElementsByTagName("subElementName") //returns a list of sub-elements of specified name  node.getChildNodes()                         //returns a list of all child nodes |

## **DOM interfaces**

The DOM defines several Java interfaces. Here are the most common interfaces:

* **Node** - The base datatype of the DOM.
* **Element** - The vast majority of the objects you'll deal with are Elements.
* **Attr** Represents an attribute of an element.
* **Text** The actual content of an Element or Attr.
* **Document** Represents the entire XML document. A Document object is often referred to as a DOM tree.

## **Common DOM methods**

When you are working with the DOM, there are several methods you'll use often:

* **Document.getDocumentElement()** - Returns the root element of the document.
* **Node.getFirstChild()** - Returns the first child of a given Node.
* **Node.getLastChild()** - Returns the last child of a given Node.
* **Node.getNextSibling()** - These methods return the next sibling of a given Node.
* **Node.getPreviousSibling()** - These methods return the previous sibling of a given Node.
* **Node.getAttribute(attrName)** - For a given Node, returns the attribute with the requested name.

**JAXB**

 For:   Use:

& &amp;

< &lt;

>&gt;

[ &#91;

] &#93;

XML Entity Names for Restricted Characters Use For & & < < > > " "





TODO : 1. One more example with attribute

1. In detail all DTD tags
2. How to use DTD to validate the XML file

XML became a W3C Recommendation on February 10, 1998.

XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

Several Internet languages are written in XML. Here are some examples:

* XHTML
* XML Schema
* SVG
* WSDL
* RSS

Tree structure:

<root>  
  <child>  
    <subchild>.....</subchild>  
  </child>  
</root>

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters).

Example:



<bookstore>  
  <book category="COOKING">  
    <title lang="en">Everyday Italian</title>  
    <author>Giada De Laurentiis</author>  
    <year>2005</year>  
    <price>30.00</price>  
  </book>  
  <book category="CHILDREN">  
    <title lang="en">Harry Potter</title>  
    <author>J K. Rowling</author>  
    <year>2005</year>  
    <price>29.99</price>  
  </book>  
  <book category="WEB">  
    <title lang="en">Learning XML</title>  
    <author>Erik T. Ray</author>  
    <year>2003</year>  
    <price>39.95</price>  
  </book>  
</bookstore>

The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>.

The <book> element has 4 children: <title>,< author>, <year>, <price>.

Meaning of eXtensible:

XML elements can be extended to carry more information.

Look at the following XML example:

<note>  
  <to>Tove</to>  
  <from>Jani</from>  
  <body>Don't forget me this weekend!</body>  
</note>

Let's imagine that we created an application that extracted the <to>, <from>, and <body> elements from the XML document to produce this output:

|  |
| --- |
| **MESSAGE**  **To:** Tove **From:** Jani  Don't forget me this weekend! |

Imagine that the author of the XML document added some extra information to it:

<note>  
  <date>2008-01-10</date>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>

Should the application break or crash?

No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output.

One of the beauties of XML, is that it can be extended without breaking applications.

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

XML elements must follow these naming rules:

1. Element names are case-sensitive
2. Element names must start with a letter or underscore
3. Element names cannot start with the letters xml (or XML, or Xml, etc)
4. Element names can contain letters, digits, hyphens, underscores, and periods
5. Element names cannot contain spaces

Any name can be used, no words are reserved (except xml).

**Naming Styles**

There are no naming styles defined for XML elements. But here are some commonly used:

|  |  |  |
| --- | --- | --- |
| **Style** | **Example** | **Description** |
| Lower case | <firstname> | All letters lower case |
| Upper case | <FIRSTNAME> | All letters upper case |
| Underscore | <first\_name> | Underscore separates words |
| Pascal case | <FirstName> | Uppercase first letter in each word |
| Camel case | <firstName> | Uppercase first letter in each word except the first |

If you choose a naming style, it is good to be consistent!

XML Attributes:

XML elements can have attributes, just like HTML.

Attributes provide additional information about an element.

* XML Attributes must be quoted

<person gender="female">

Or,

<person gender='female'>

<gangster name='George "Shotgun" Ziegler'>

Or you can use character entities:

<gangster name="George &quot;Shotgun&quot; Ziegler">

## **XML Elements vs. Attributes**

Take a look at these examples:

<person gender="female">  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>

<person>  
  <gender>female</gender>  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>

In the first example gender is an attribute. In the last, gender is an element. Both examples provide the same information.

There are no rules about when to use attributes or when to use elements. Attributes are handy in HTML. In XML my advice is to avoid them. Use elements instead.

## **My Favorite Way**

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

<note date="2008-01-10">  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>

A date element is used in the second example:

<note>  
  <date>2008-01-10</date>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>

An expanded date element is used in the third: (THIS IS MY FAVORITE):

<note>  
  <date>  
    <year>2008</year>  
    <month>01</month>  
    <day>10</day>  
  </date>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>

## **Avoid XML Attributes?**

Some of the problems with using attributes are:

* attributes cannot contain multiple values (elements can)
* attributes cannot contain tree structures (elements can)
* attributes are not easily expandable (for future changes)

Attributes are difficult to read and maintain. Use elements for data. Use attributes for information that is not relevant to the data.

Don't end up like this:

<note day="10" month="01" year="2008"  
to="Tove" from="Jani" heading="Reminder"  
body="Don't forget me this weekend!">  
</note>

XML Namespaces provide a method to avoid element name conflicts.

## **XML Namespaces - The xmlns Attribute**

When using prefixes in XML, a so-called **namespace** for the prefix must be defined.

The namespace is defined by the **xmlns attribute** in the start tag of an element.

The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

<root>  
  
<h:table xmlns:h="http://www.w3.org/TR/html4/">  
  <h:tr>  
    <h:td>Apples</h:td>  
    <h:td>Bananas</h:td>  
  </h:tr>  
</h:table>  
  
<f:table xmlns:f="http://www.w3schools.com/furniture">  
  <f:name>African Coffee Table</f:name>  
  <f:width>80</f:width>  
  <f:length>120</f:length>  
</f:table>  
  
</root>

In the example above, the xmlns attribute in the <table> tag give the h: and f: prefixes a qualified namespace.

When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

Namespaces can be declared in the elements where they are used or in the XML root element:

<root xmlns:h="http://www.w3.org/TR/html4/"  
xmlns:f="http://www.w3schools.com/furniture">  
  
<h:table>  
  <h:tr>  
    <h:td>Apples</h:td>  
    <h:td>Bananas</h:td>  
  </h:tr>  
</h:table>  
  
<f:table>  
  <f:name>African Coffee Table</f:name>  
  <f:width>80</f:width>  
  <f:length>120</f:length>  
</f:table>  
  
</root>

**Note:** The namespace URI is not used by the parser to look up information.

The purpose is to give the namespace a unique name. However, often companies use the namespace as a pointer to a web page containing namespace information.

## **Uniform Resource Identifier (URI)**

A **Uniform Resource Identifier** (URI) is a string of characters which identifies an Internet Resource.

The most common URI is the **Uniform Resource Locator** (URL) which identifies an Internet domain address. Another, not so common type of URI is the **Universal Resource Name** (URN).

In our examples we will only use URLs.

## **Default Namespaces**

Defining a default namespace for an element saves us from using prefixes in all the child elements. It has the following syntax:

xmlns="*namespaceURI*"

This XML carries HTML table information:

<table xmlns="http://www.w3.org/TR/html4/">  
  <tr>  
    <td>Apples</td>  
    <td>Bananas</td>  
  </tr>  
</table>

This XML carries information about a piece of furniture:

<table xmlns="http://www.w3schools.com/furniture">  
  <name>African Coffee Table</name>  
  <width>80</width>  
  <length>120</length>  
</table>

## **Namespaces in Real Use**

XSLT is an XML language that can be used to transform XML documents into other formats, like HTML.

In the XSLT document below, you can see that most of the tags are HTML tags.

The tags that are not HTML tags have the prefix xsl, identified by the namespace xmlns:xsl="http://www.w3.org/1999/XSL/Transform":

<?xml version="1.0" encoding="UTF-8**"**?>  
  
<xsl:stylesheet version="1.0"  
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
  
<xsl:template match="/">  
<html>  
<body>  
  <h2>My CD Collection</h2>  
  <table border="1">  
    <tr>  
      <th style="text-align:left">Title</th>  
      <th style="text-align:left">Artist</th>  
    </tr>  
    <xsl:for-each select="catalog/cd">  
    <tr>  
      <td><xsl:value-of select="title"/></td>  
      <td><xsl:value-of select="artist"/></td>  
    </tr>  
    </xsl:for-each>  
  </table>  
</body>  
</html>  
</xsl:template>  
  
</xsl:stylesheet>

XML Encoding:

XML documents can contain international characters, like Norwegian æøå, or French êèé.

To avoid errors, you should specify the encoding used, or save your XML files as UTF-8.

## **Character Encoding**

Character encoding defines a unique binary code for each different character used in a document.

In computer terms, character encoding are also called character set, character map, code set, and code page.

## **The Unicode Consortium**

The Unicode Consortium develops the Unicode Standard. Their goal is to replace the existing character sets with its standard Unicode Transformation Format (UTF).

The Unicode Standard has become a success and is implemented in HTML, XML, Java, JavaScript, E-mail, ASP, PHP, etc. The Unicode standard is also supported in many operating systems and all modern browsers.

The Unicode Consortium cooperates with the leading standards development organizations, like ISO, W3C, and ECMA.

## **The Unicode Character Sets**

Unicode can be implemented by different character sets. The most commonly used encodings are UTF-8 and UTF-16.

UTF-8 uses 1 byte (8-bits) to represent basic Latin characters, and two, three, or four bytes for the rest.

UTF-16 uses 2 bytes (16 bits) for most characters, and four bytes for the rest.

## **UTF-8 = The Web Standard**

UTF-8 is the standard character encoding on the web.

UTF-8 is the default character encoding for HTML5, CSS, JavaScript, PHP, SQL, and XML.

## **XML Encoding**

The ﬁrst line in an XML document is called the **prolog**:

<?xml version="1.0"?>

The prolog is optional. Normally it contains the XML version number.

It can also contain information about the encoding used in the document. This prolog specifies UTF-8 encoding:

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

The XML standard states that all XML software must understand both UTF-8 and UTF-16.

UTF-8 is the default for documents without encoding information.

In addition, most XML software systems understand encodings like ISO-8859-1, Windows-1252, and ASCII.

## **XML Errors**

Most often, XML documents are created on one computer, uploaded to a server on a second computer, and displayed by a browser on a third computer.

If the encoding is not correctly interpreted by all the three computers, the browser might display meaningless text, or you might get an error message.

For high quality XML documents, UTF-8 encoding is the best to use. UTF-8 covers international characters, and it is also the default, if no encoding is declared.

## **Conclusion**

When you write an XML document:

* Use an XML editor that supports encoding
* Make sure you know what encoding the editor uses
* Describe the encoding in the encoding attribute
* UTF-8 is the safest encoding to use
* UTF-8 is the web standard

# **XML Comments**

XML comments are just like HTML comments. We know that the comments are used to make codes more understandable other developers.

XML Comments add notes or lines for understanding the purpose of an XML code. Although XML is known as self-describing data but sometimes XML comments are necessary.

#### Syntax

An XML comment should be written as:

1. <!-- Write your comment-->

## **Rules for adding XML comments**

* Don't use a comment before an XML declaration.
* You can use a comment anywhere in XML document except within attribute value.
* Don't nest a comment inside the other comment.

## **XML Validation**

XML file can be validated by 2 ways:

1. against DTD
2. against XSD

DTD (Document Type Definition) and XSD (XML Schema Definition) are used to define XML structure.

## **XML DTD**

A DTD defines the legal elements of an XML document

In simple words we can say that a DTD defines the document structure with a list of legal elements and attributes.

XML schema is a XML based alternative to DTD.

Actually DTD and XML schema both are used to form a well formed XML document.

We should avoid errors in XML documents because they will stop the XML programs.

## **XML schema**

It is defined as an XML language

Uses namespaces to allow for reuses of existing definitions

It supports a large number of built in data types and definition of derived data types

For the newbies, SAX stands for **Simple API for XML** and DOM stands for **Document Object Model**.  
Both of these are APIs to parse XML documents. There is no direct competition between DOM and SAX  
as both of them have there own strengths and weaknesses.

The following are some fundamental differnces between a SAX and DOM model of parsing.

**SAX:**

1. Parses the document on node by node basis.  
2. Does not store the XML in memory.  
3. We cannot insert or delete a node.  
4. This model uses top to bottom traversing.  
5. This model does not preserve comments.  
6. It runs little faster than DOM

**DOM**

1. Stores the entire XML document into memory before processing  
2. Occupies more memory  
3. We can insert or delete nodes  
4. This model can traverse in any direction.  
5. This model preserves comments.  
6. It runs slower than SAX model

So, when to choose what model to use?

Here is what you can do.

If you just need to read a node, but do not require to insert/delte node, then use SAX.  
If you require node manipulation (insert/delete) nodes, use DOM.

There are various parsers in Java that can be used. The following is a list of few java parsers.

* Apache Parser [Xerces]
* Oracle XML Parser
* Sun XML Parser

All of these parsers provide SAX and DOM parsing mechanisms.

Element, tag, node

* An element consists of an opening tag, its attributes, any content, and a closing tag.
* A tag – either opening or closing – is used to mark the start or end of an element.
* A node is a part of the hierarchical structure that makes up an XML document. “Node” is a generic term that applies to any type of XML document object, including elements, attributes, comments, processing instructions, and plain text.

**Main XML Related Technologies**

|  |  |  |
| --- | --- | --- |
| Structure Definitions | DTD | Schema definition language defined under the XML 1.0 specification |
| XML Schema | Schema definition language defining XML document structure |
| Selection Method | XPath | Language designating syntax that indicates the appropriate path within a tree structure in order to select elements and/ or attributes within an XML document |
| Transformation | XSLT | Language specifying the mechanism for transforming an XML document into an XML document of different structure or structured data |
| Display | CSS | A specification defining the layout for XML and HTML displayed through a Web browser |
| XSL-FO | Language for describing the layout of an XML document |
| Protocol | SOAP | XML-based messaging protocol |

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Technology** | **Meaning** | **Description** |
| 1) | XHTML | Extensible html | It is a clearer and stricter version of XML. It belongs to the family of XML markup languages. It was developed to make html more extensible and increase inter-operability with other data. |
| 2) | XML DOM | XML document object model | It is a standard document model that is used to access and manipulate XML. It defines the XML file in tree structure. |
| 3) | XSL it contain three parts: i) XSLT (xsl transform) ii) XSL iii)XPath | Extensible style sheet language | i) It transforms XML into other formats, like html. ii) It is used for formatting XML to screen, paper etc. iii) It is a language to navigate XML documents. |
| 4) | XQuery | XML query language | It is a XML based language which is used to query XML based data. |
| 5) | DTD | Document type definition | It is an standard which is used to define the legal elements in an XML document. |
| 6) | XSD | XML schema definition | It is an XML based alternative to dtd. It is used to describe the structure of an XML document. |
| 7) | XLink | XML linking language | xlink stands for XML linking language. This is a language for creating hyperlinks (external and internal links) in XML documents. |
| 8) | XPointer | XML pointer language | It is a system for addressing components of XML based internet media. It allows the xlink hyperlinks to point to more specific parts in the XML document. |
| 9) | SOAP | Simple object access protocol | It is an acronym stands simple object access protocol. It is XML based protocol to let applications exchange information over http. in simple words you can say that it is protocol used for accessing web services. |
| 10) | WSDL | web services description languages | It is an XML based language to describe web services. It also describes the functionality offered by a web service. |
| 11) | RDF | Resource description framework | RDF is an XML based language to describe web resources. It is a standard model for data interchange on the web. It is used to describe the title, author, content and copyright information of a web page. |
| 12) | SVG | Scalable vector graphics | It is an XML based vector image format for two-dimensional images. It defines graphics in XML format. It also supports animation. |
| 13) | RSS | Really simple syndication | RSS is a XML-based format to handle web content syndication. It is used for fast browsing for news and updates. It is generally used for news like sites. |

|  |
| --- |
| **XML Related Technologies** Here we have pointed out XML related technologies. There are following XML related technologies: |

# The standalone pseudo-attribute is only relevant if a DTD is used

The standalone pseudo-attribute in the XML declaration is a mystery for many XML beginners. Most often it is irrelevant but it is tempting to add it or delete it or to change its value from "yes" to "no" to "yes" as long as you don't know what it is all about.

* 1. [Standalone="no"](http://www.xmlplease.com/xml/xmlquotations/standalone#s1.)
* 2. [Default values and Entities](http://www.xmlplease.com/xml/xmlquotations/standalone#s2.)
* 3. [When an XML document can "standalone"](http://www.xmlplease.com/xml/xmlquotations/standalone#s3.)
* 4. [Standalone="yes"](http://www.xmlplease.com/xml/xmlquotations/standalone#s4.)
* 5. [What about XML Schema](http://www.xmlplease.com/xml/xmlquotations/standalone#s5.)

The XML declaration must have a "version" pseudo-attribute, but can also have an "encoding" and a "standalone" pseudo-attribute. The "standalone" pseudo-attribute is defined in a section called [Standalone Document Declaration](http://www.w3.org/TR/2004/REC-xml-20040204/#sec-rmd) in the XML Recommendation (standard). [[1]](http://www.xmlplease.com/xml/xmlquotations/standalone#footnote-1)

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

Let us be more precise. A DTD can consist of an internal DTD subset declared in the top of the XML document inside the DTD declaration and/or an external DTD subset declared in a separate text document. Most often a DTD only consists of a DTD declaration in the top of an XML document, right after the XML declaration, pointing to the external DTD subset. The standalone pseudo-attribute could only be relevant if an external DTD subset exists.

The "standalone"-attribute is completely irrelevant if an XML document is not using a DTD. This is most often the case today. Most XML documents use an XML Schema schema instead of a DTD.

Since most XML documents don't use DTDs, it is pretty safe to make it into a rule of thumb for beginners: "Just don't bother about the standalone pseudo-attribute. If it shows up just ignore it, or delete it, and it does not matter if the value is "yes" or "no".

## 1. Standalone="no"

Only if the XML document has a DTD, the "standalone" pseudo-attribute could be of importance. If the XML declaration has no "standalone" pseudo-attribute it defaults to standalone="no". It is never necessary to use standalone="no" explicitly.

If the standalone pseudo-attribute is not used or if it is used explicitly with the value "no", it is a signal to the XML processor that the DTD is not just for validation: there could be things to look up in the DTD like:

* Default values for attributes
* Entity declarations

The two items above are the important things to remember. But the XML spec also talks about normalization of attribute values and about ignorable whitespace in elements containing elements only. If e.g. line breaks are used in an attribute value or whitespace exist in an element containing elements only, the XML processor normalizes the document removing such whitespace.

The XML declaration must implicitly or explicitly use standalone="no" if the DTD effects the XML document with default values for attributes or "external markup declaration" (entity declarations), or if the DTD is needed to normalize attribute values or to remove ignorable whitespace from element content.

## 2. Default values and Entities

Default values for attributes we also know from XML Schema but many XML developers don't know what "external markup declarations" are all about, something specific to the use of DTDs.

In a DTD we can declare that an entity like &jt; should be replaced by "Jesper Tverskov" whenever encountered in the XML instance document. In the DTD the declaration looks like this:

<!ENTITY jt "Jesper Tverskov">

In much the same way we can make up entities that can include XML markup or text placed in external files. Entities for external markup declarations and text are only possible when we use DTDs. We have nothing similar in XML Schema. [[2]](http://www.xmlplease.com/xml/xmlquotations/standalone#footnote-2)

## 3. When an XML document can "standalone"

If there are no default values for attributes declared in the DTD and no external entities are declared in the DTD, and if no normalization is needed, the XML document can "standalone", that is the DTD is only for validation.

Only if the XML document can stand alone in the "DTD" sense, could it be relevant to use the standalone pseudo-attribute with the value "yes".

Even if the XML document can stand alone, there is often no need to use the standalone pseudo-attribute. It depends on the situation, on the software, XML-processor and network involved.

If the DTD is small and the external DTD subset file (\*.dtd) is on the same server, or if things look to work well and fast, we don't have a problem, and we don't need to use the standalone pseudo-attribute.

## 4. Standalone="yes"

If the XML document has a DTD and standalone="yes" is used, it is a signal to the XML processor that the DTD is only for validation. The DTD is not needed to look anything up or to normalize attribute values or to remove ignorable whitespace from element content.

This is a useful option to consider if things work slower than we would like them to do. If the DTD is huge or the network slow and the DTD is only used for validation, then and only then is it relevant to use the standalone pseudo-attribute.

If our XML document has an external DTD subset and can not stand alone, and the XML document loads slowly, we should probably reconsider our project. Why not remake our DTD to get rid of externally declared entities and of default values for attributes and why not normalize attribute values ourselves and remove ignorable whitespace from element content in order to make it possible to use standalone="yes" to make our document load faster?

## 5. What about XML Schema

We can not declare external entities in XML Schema or in RELAX NG for that matter. This is only an option in a DTD. But in XML Schema we can also have default values for attributes like in a DTD. So why does the standalone pseudo-attribute not apply also to XML Schema?

For a very obvious reason: DTD is defined in the XML Recommendation (standard). It is only natural that a pseudo-attribute in the XML declaration can deal with a DTD problem. The XML Recommendation on the other hand knows today nothing about XML Schema or other schema languages being independent recommendations or standards made years after the XML Recommendation.

## Footnotes

[[1](http://www.xmlplease.com/xml/xmlquotations/standalone#footnote-1-referrer)]

"Version", "encoding" and "standalone" look like attributes and is doing a job similar to attributes, but only elements can have attributes. The order of real attributes are not important, and we can get to real attributes with XPath expressions like @\*, etc.

The order of the pseudo-attributes is important on the other hand, their sequence must be "version", "encoding", "standalone" if they all exist, and in XPath 1.0 we have no expressions or functions that can read or detect if an XML declaration is present or the values of the pseudo-attributes.

In XPath 2.0 we can get to the pseudo-attributes, but it is not easy. See my article: [Using unparsed-text() in XSLT 2.0 to test prolog](http://www.xmltraining.biz/prolog).

[[2](http://www.xmlplease.com/xml/xmlquotations/standalone#footnote-2-referrer)]

A schema language like RELAX NG has even made a point out of this: A schema should only be for validation. It is not the job of a RELAX NG schema to change or modify the XML instance document supplying externally declared markup or text for entity references or default values for attributes.

***XML Declaration***

The XML declaration is a [processing instruction](http://xmlwriter.net/xml_guide/processing_instruction.shtml) that identifies the document as being XML. All XML documents should begin with an XML declaration.

|  |
| --- |
| <?xml  [version](http://xmlwriter.net/xml_guide/xml_declaration.shtml#Version)="version\_number"  [encoding](http://xmlwriter.net/xml_guide/xml_declaration.shtml#Encoding)="encoding\_declaration"  [standalone](http://xmlwriter.net/xml_guide/xml_declaration.shtml#Standalone)="standalone\_status" ?> |

|  |
| --- |
| **Example:** |
| <?xml version="1.0" encoding="UTF-8" standalone="no" ?> |

**Rules:**

* If the XML declaration is included, it must be situated at the first position of the first line in the XML documentwell-formedness constraint.
* If the XML declaration is included, it must contain the [version](http://xmlwriter.net/xml_guide/xml_declaration.shtml#Version) number attributewell-formedness constraint.
* If all of the attributes[glossary](http://xmlwriter.net/xml_guide/glossary.shtml#Attribute) are declared in an XML declaration, they must be placed in the order shown abovewell-formedness constraint.
* If any [elements](http://xmlwriter.net/xml_guide/element_declaration.shtml), [attributes](http://xmlwriter.net/xml_guide/attlist_declaration.shtml), or [entities](http://xmlwriter.net/xml_guide/entity_declaration.shtml) are used in the XML document that are referenced or defined in an [external DTD](http://xmlwriter.net/xml_guide/doctype_declaration.shtml#ExternalDTD), standalone="no"must be includedvalidity constraint.
* The XML declaration must be in lower case (except for the encoding declarations)well-formedness constraint.

**Note:**

* The XML declaration has no closing tag, that is </?xml>well-formedness constraint.

The following table shows a list of the possible attributes that may be used in the XML declaration.

|  |  |  |
| --- | --- | --- |
| **Attribute Name:** | **Possible Attribute Value:** | **Attribute Description:** |
| version | 1.0 | Specifies the version of the XML standard that the XML document conforms to. The version attribute must be included if the XML declaration is declaredwell-formedness constraint. |
| encoding | UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift\_JIS, EUC-JP | These are the encoding names[glossary](http://xmlwriter.net/xml_guide/glossary.shtml#EncodingNames) of the most common character sets in use today. For a full list of encodings check the IANA's[glossary](http://xmlwriter.net/xml_guide/glossary.shtml#IANA) website. |
| standalone | yes, no | Use 'yes' if the XML document has an [internal DTD](http://xmlwriter.net/xml_guide/doctype_declaration.shtml#internalDTD). Use 'no' if the XML document is linked to an [external DTD](http://xmlwriter.net/xml_guide/doctype_declaration.shtml#externalDTD), or any external [entity](http://xmlwriter.net/xml_guide/entity_declaration.shtml)referencesvalidity constraint. |

### Entity declarations[[edit](http://en.wikipedia.org/w/index.php?title=Document_type_definition&action=edit&section=6)]

An entity is similar to a [macro](http://en.wikipedia.org/wiki/Macro_(computer_science)). The entity declaration assigns it a value that is retained throughout the document. A common use is to have a name more recognizable than a numeric character reference for an unfamiliar character.[[5]](http://en.wikipedia.org/wiki/Document_type_definition#cite_note-5) Entities help to improve legibility of an XML text. In general, there are two types: internal and external.

* **Internal (parsed) entities** are associating a name with any arbitrary textual content defined in their declaration (which may be in the *internal subset* or in the *external subset*of the DTD declared in the document). When a named entity reference is then encountered in the rest of the document (including in the rest of the DTD), and if this entity name has effectively been defined as a parsed entity, the reference itself is replaced immediately by the textual content defined in the parsed entity, and the parsing continues within this replacement text.
  + **Predefined named character entities** are similar to internal entities: 5 of them however are treated specially in all SGML, HTML and XML parsers. These entities are a bit different from normal parsed entities, because when a named character entity reference is encountered in the document, the reference is also replaced immediately by the character content defined in the entity, but the parsing continues **after** the replacement text, which is immediately inserted literally in the currently parsed token (if such character is permitted in the textual value of that token). This allows some characters that are needed for the core syntax of HTML or XML themselves to be escaped from their special syntactic role (notably "&" which is reserved for beginning entity references, "<" or ">" which delimit the markup tags, and "double" or 'single' quotation marks, which delimit the values of attributes and entity definitions). Predefined character entities also include numeric character references that are handled the same way and can also be used to escape the characters they represent, or to bypass limitations in the character repertoire supported by the document encoding.
  + In basic profiles for SGML or in HTML documents, the declaration of internal entities is not possible (because external DTD subsets are not retrieved, and internal DTD subsets are not supported in these basic profiles).
  + Instead, HTML standards predefine a large set of several hundred named character entities, which can still be handled as standard parsed entities defined in the DTD used by the parser.
* **External entities** refer to external storage objects. They are just declared by a unique name in the document, and defined with a public identifier (an FPI) and/or a system identifier (interpreted as an [URI](http://en.wikipedia.org/wiki/URI)) specifying where the source of their content. They exist in fact in two variants:
  + **parsed external entities** (most often defined with a SYSTEM identifier indicating the URI of their content) that are *not* associated in their definition to a named annotation, in which case validating XML or SGML parsers retrieve their contents and parse them as if they were declared as internal entities (the external entity containing their effective replacement text);
  + **unparsed external entities** that are defined and associated with an annotation name, in which case they are treated as opaque references and signaled as such to the application using the SGML or XML parser: their interpretation, retrieval and parsing is left to the application, according the types of annotations it supports (see the next section about annotations and for examples of unparsed external entities).
  + External entities are not supported in basic profiles for SGML or in HTML documents, but are valid in full implementations of SGML and in XML 1.0 or 1.1 (including XHTML and SVG, even if they are not strictly needed in those document types).

An example of internal entity declarations (here in an internal DTD subset of an SGML document) is:

<!DOCTYPE sgml [

<!ELEMENT sgml ANY>

<!ENTITY % std "standard SGML"**>**

<!ENTITY % signature " &#x2014; &author;."**>**

<!ENTITY % question "Why couldn&#x2019;t I publish my books directly in %std;?"**>**

<!ENTITY % author "William Shakespeare"**>**

]>

**<sgml>**&question;&signature;**</sgml>**

Note that internal entities may be defined in any order, as long as they are not referenced and parsed in the DTD or in the body of the document, in their order of parsing: it is valid to include a reference to a still undefined entity within the content of a parsed entity, but it is invalid to include anywhere else any named entity reference before this entity has been fully defined, including all other internal entities referenced in its defined content (this also prevents circular or recursive definitions of internal entities). This document is parsed as if it was:

<!DOCTYPE sgml [

<!ELEMENT sgml ANY>

<!ENTITY % std "standard SGML"**>**

<!ENTITY % signature " — &author;."**>**

<!ENTITY % question "Why couldn’t I publish my books directly in standard SGML?"**>**

<!ENTITY % author "William Shakespeare"**>**

]>

**<sgml>**Why couldn’t I publish my books directly in standard SGML? — William Shakespeare.**</sgml>**

Note that reference to the "author" internal entity is not substituted in the replacement text of the "signature" internal entity. Instead, it is replaced only when the "signature" entity reference is parsed within the content of the "sgml" element, but only by validating parsers (non-validating parsers do not substitute entity references occurring within contents of element or within attribute values, in the body of the document.

This is possible because the replacement text specified in the internal entity definitions permits a distinction between **parameter** entity references (that are introduced by the "%" character and whose replacement applies to the parsed DTD contents) and **general** entity references (that are introduced by the "&" character and whose replacement is delayed until they are effectively parsed and validated). The "%" character for introducing parameter entity references in the DTD loses its special role outside the DTD and it becomes a literal character.

However, the references to predefined numeric character entities are substituted wherever they occur, without needing a validating parser (they are only introduced by the "&" character).

PUBLIC vs SYSTEM

If the XML document type declaration includes any SYSTEM identifier for the external subset, it can't be safely processed as standalone: the URI should be retrieved, otherwise there may be unknown named character entities whose definition may be needed to correctly parse the effective XML syntax in the internal subset or in the document body (the XML syntax parsing is normally performed *after* the substitution of all named entities, excluding the five entities that are predefined in XML and that are implicitly substituted *after*parsing the XML document into lexical tokens). If it just includes any PUBLIC identifier, it *may* be processed as standalone, if the XML processor knows this PUBLIC identifier in its local catalog from where it can retrieve an associated DTD entity.

**Document type declaration**  
The document type declaration tells you and the parser which DTD (document type definition) governs the current document. Usually the DTD is incorporated by reference to a PUBLIC or SYSTEM name by which the parser or SGML application can find the DTD.   
The declaration portion of an SGML document will also declare document-specific components:   
- Entities used in the document (also addressable as PUBLIC or SYSTEM entities);   
- Any notation types not declared in the main DTD   
...  
**Keywords PUBLIC and SYSTEM**  
PUBLIC entities are those entities assumed to be known to many systems so that a full declaration need not be transmitted. This format is generally used to declare publicly available DTDs, standard character sets, and commonly used notations such as TIFF.   
A formal public identifier must also include:   
- the "owner" (usually the originator) of the entity, such as the ISO, Department of Defense, or a publishing house.   
- the public text class, such as ENTITY, DTD, SUBDOC, TEXT, or NOTATION.   
- public language code, such as EN for English.   
SYSTEM entities are not assumed to be known to a receiving system. Thus, such entities require a full declaration of system identification (path, etc.) when they are exchanged

The SYSTEM is used to specify the location of a DTD. The PUBLIC keyword is used for the same purpose.  
You would use SYSTEM when the DTD is to be used between multiple organizations. The PUBLIC keyword is used when the DTD is used internally and when the location of the DTD is internal to the organization. PUBLIc is also used in legacy SGML applications.  
They can also be combined together, in this case if the application cannot locate the DTD using the PUBLIC location then the SYSTEM location is used.

The DOCTYPE tag occurs after the XML declaration and before the root element.  
The SYSTEM identifier specifies the location of the DTD file. Since it does not  
start with a prefix like http:/or file:/, the path is relative to the location of  
the XML document. The parser is using that information to find the DTD file, just as your application would to  
find a file relative to the XML document.

First of all, make sure that your XML parser supports schema validation. Not all do, yet.

Note that by adapting the first *setAttribute* call this code can also validate against a DTD or Relax-NG schema. Consult [JavaDoc:javax.xml.XMLConstants](http://docs.oracle.com/javase/8/docs/api/javax/xml/XMLConstants.html) for the corresponding attribute value.

Here's a quick sample for schema validation using JAXP 1.2.

import java.io.File;  
import java.io.IOException;  
  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.DocumentBuilderFactory;  
import javax.xml.parsers.ParserConfigurationException;  
  
import org.w3c.dom.Document;  
import org.xml.sax.SAXException;  
   
public class SchemaValidationExample {  
   
    public static void main(String args[]) {  
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
  
        factory.setValidating(true);  
  
    factory.setAttribute(  
          "http://java.sun.com/xml/jaxp/properties/schemaLanguage",   
          "http://www.w3.org/2001/XMLSchema");  
    factory.setAttribute(  
      "http://java.sun.com/xml/jaxp/properties/schemaSource",  
      "http://domain.com/mynamespace/mySchema.xsd");  
    Document doc = null;  
    try{          
         DocumentBuilder parser = factory.newDocumentBuilder();  
         doc = parser.parse("data.xml");  
       }  
    catch (ParserConfigurationException e){  
         System.out.println("Parser not configured: " + e.getMessage());  
       }  
    catch (SAXException e){  
         System.out.print("Parsing XML failed due to a " + e.getClass().getName() + ":");  
         System.out.println(e.getMessage());  
       }  
    catch (IOException e){  
         e.printStackTrace();  
       }  
    }  
}