

# Haute Ecole de la Province de Liège

## Catégorie Technique

### XML Basics

Technologie de l'e-commerce - Partim Théorie 1

Ludovic Kutý

Informatique de Gestion  
3<sup>èmes</sup> Bachelier  
Groupes 230x

**Siège social :**  
Avenue Montesquieu, 6  
4101 Jemeppe (Seraing)  
Belgique

Année Académique 2011–2012



## **Catégorie Technique**

### **XML Basics**

Technologie de l'e-commerce - Partim Théorie 1

Ludovic Kutu

Informatique de Gestion  
3<sup>èmes</sup> Bachelier  
Groupes 230x

Année Académique 2011–2012



# Contents

<b>1</b>	<b>Extensible Markup Language (XML)</b>	<b>7</b>
1.1	Presentation . . . . .	7
1.1.1	Meaning with tags . . . . .	8
1.1.2	XML ecosystem . . . . .	10
1.1.3	Data flow . . . . .	10
1.2	Physical and logical structures . . . . .	11
1.2.1	Physical structure . . . . .	11
1.2.2	Logical structure . . . . .	13
1.3	XML constituents . . . . .	14
1.3.1	Root element . . . . .	14
1.3.2	XML declaration . . . . .	15
1.3.3	Characters . . . . .	15
1.3.4	Processing instructions . . . . .	16
1.3.5	Elements . . . . .	16
1.3.6	Attributes . . . . .	17
1.3.7	Elements vs attributes . . . . .	18
1.3.8	Character references . . . . .	18
1.3.9	CDATA sections . . . . .	19
1.3.10	Comments . . . . .	19
1.4	Tree structure . . . . .	19
1.5	Well-formedness . . . . .	21
1.6	Translations . . . . .	22
<b>2</b>	<b>Schema Languages and Schemas</b>	<b>23</b>
2.1	Schemas . . . . .	23
2.2	Schema languages . . . . .	24
2.3	Schema instances and validity . . . . .	26
<b>3</b>	<b>Document Type Definition (DTD)</b>	<b>29</b>
3.1	Presentation . . . . .	29
3.2	Declarations . . . . .	29
3.2.1	Internal declaration . . . . .	30
3.2.2	External system declaration . . . . .	30
3.2.3	External public declaration . . . . .	31
3.2.4	External and internal declarations . . . . .	31
3.3	Catalogs . . . . .	32
3.4	Syntax . . . . .	35
3.4.1	Elements . . . . .	35
3.4.2	Attributes . . . . .	37
3.5	Example: book document . . . . .	41
3.6	Entities . . . . .	41
3.6.1	General entities . . . . .	42
3.6.2	Parameter entities . . . . .	43
3.7	Validity . . . . .	45
3.8	Translations . . . . .	46

<b>4</b>	<b>Namespaces in XML</b>	<b>47</b>
4.1	Presentation . . . . .	47
4.2	Expanded names . . . . .	48
4.3	Qualified names . . . . .	48
4.4	Declaration and binding . . . . .	49
4.5	XML sample document with prefixes . . . . .	50
4.6	Default namespace . . . . .	51
4.7	Fixing the NS problem in DTD . . . . .	52
<b>5</b>	<b>APIs and Implementations</b>	<b>55</b>
5.1	APIs . . . . .	55
5.1.1	Event-based APIs . . . . .	56
5.1.2	Tree-based APIs . . . . .	57
5.1.3	Data-binding APIs . . . . .	57
5.2	Implementations . . . . .	57
<b>6</b>	<b>Simple API for XML (SAX)</b>	<b>59</b>
6.1	Presentation . . . . .	59
6.2	Java packages and documentation . . . . .	60
6.3	Example 1: parsing does nothing . . . . .	60
6.3.1	Parser instantiation . . . . .	61
6.3.2	Register callback methods . . . . .	62
6.3.3	Parsing . . . . .	62
6.3.4	Exceptions . . . . .	63
6.4	Features and properties . . . . .	63
6.5	Example 2: display data . . . . .	64
6.5.1	ContentHandler startElement . . . . .	65
6.5.2	ContentHandler endElement . . . . .	66
6.5.3	ContentHandler characters . . . . .	67
6.5.4	ContentHandler ignorableWhitespace . . . . .	67
6.6	Example 3: handling errors . . . . .	68

# Chapter 1

## Extensible Markup Language (XML)

### 1.1 Presentation

#### versions

---

- XML is a W3C recommendation
- Version 1.0
  - 5th edition
  - since 2008/11/26
  - <http://www.w3.org/TR/xml/>
- Version 1.1
  - 2nd edition
  - since 2006/09/29
  - <http://www.w3.org/TR/xml11/>

#### definition

---

XML = eXtensible Markup Language

- Meta-language with minimal syntax  $\Rightarrow$  used to define XML applications or vocabularies
- Gives meaning to data with tags
- Well defined basic syntax but no restriction on tags and structure
- The term "applications" does not mean "programs" in this context. It just means a specific use of the XML language to define a new markup vocabulary.
- The meaning is somewhat understandable by a human being but, of course, not by a computer unless the program reading the XML document has been conceived to know those exact tags.

A sample document might be:

```
1 <livre>
2   <titre>Les écureuils de Central Park sont tristes le lundi</titre>
3   <auteur>
4     <nom>Pancol</nom>
5     <prénom>Katherine</prénom>
6   </auteur>
7 </livre>
```

or

```

1 <book>
2   <title>Les écureuils de Central Park sont tristes le lundi</title>
3   <author>
4     <lastname>Pancol</lastname>
5     <firstname>Katherine</firstname>
6   </author>
7 </book>

```

We chose freely the name of the elements: livre, titre, auteur, nom, prénom or book, title, author, lastname, firstname. It was not written somewhere that we had to use them, we chose them. This is the freedom we get with XML. Note that special characters like <, > or / had to be written and that there are some syntactic rules like elements must be properly nested, the end-tag should have the same name as the start-tag, ...

### 1.1.1 Meaning with tags

#### meaning with tags

- For very simple kind of documents, tags alone are enough to understand things
- But usually we need more informations to understand/process the document:
  - Rules to constrain data: a schema
  - Documentation to understand the meaning of data: books, articles, ...
- Think about a complex MathML or SVG document
- How could you possibly understand what is going on without any visual clue ?

#### meaning with tags

#### MathML

```

<mrow> <mrow> <mrow> <mo>(</mo> <mrow> <mrow> <mn>2</mn> <mo>&#8290;</mo> <mi>k</mi>
</mrow> <mo>-</mo> <mn>1</mn> </mrow> <mo>)</mo> </mrow> <mo>!!</mo> </mrow> <mo>
&#8290;</mo> <mrow> <munderover> <mo>&#8721;</mo> <mrow> <msub> <mi>i</mi> <mn>1</
mn> </msub> <mo>=</mo> <mn>1</mn> </mrow> <mi>n</mi> </munderover>

```

This sample of a complex MathML document lets you see that without any additional documentation, it is really hard to understand what it is all about. What are mrow, mo, mi or mn ? In which context can we use them ? Could an mo be part of an mi ? All these questions have no easy answer.

#### meaning with tags

#### SVG

```

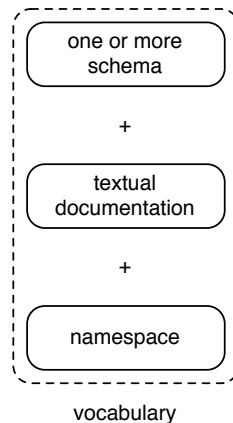
<path d="M136.128 28.837C129.728 29.637 104.999 5.605 119.328 37.637C136.128 75.193
60.394 76.482 44.128 65.637C27.328 54.437 51.328 84.037 51.328 84.037C69.728 104.037
35.328 87.237 35.328 87.237C0.928 74.437 -23.072 100.037 -26.272 100.837C-29.472
101.637 -34.272 104.837 -35.072 98.437C-35.872 92.037 -42.989 75.839 -75.073 101.637
C-98.782 120.474 -111.655 100.11 -111.655 100.11L-115.655 107.237C-137.455 70.437
-124.236 128.765 -124.236 128.765C-115.436 163.165 16.128 121.637 16.128 121.637C16
.128 121.637 184.928 91.237 196.129 87.237C207.329 83.237 299.911 89.419 299.911
89.419L294.529 71.092C229.729 24.691 212.929 49.637 199.329z"/>

```



This extract is more or less similar to the previous MathML example. We might guess that a `path` element describes a graphical path whose coordinates are given in the same element. Yes, that's true but now the question is: what does this path represent ? You can't answer that question easily. Of course we need additional documentation to use SVG but in fact, in this case, we also need a computer program to *create* the SVG document. This example shows that XML might be generated by a computer program, in this case a vector drawing program like Adobe Illustrator or Inkscape.

### vocabularies



We just saw that the XML language is a language that allows us to write other languages, that is a meta-language. The MathML and SVG vocabularies are just examples of that. There are a lot of XML vocabularies in the world of which the list on Wikipedia<sup>1</sup> is just a really short one, but that gives an idea. There are more at "XML Applications and Initiatives"<sup>2</sup>.

Since you can define a new XML vocabulary in 10 minutes, you can imagine how much vocabularies exists out there. However it is not easy to make a good vocabulary for general use. You have to design it carefully to avoid later problems. Tim Bray, one of the editors of the XML recommendation, even tell us "Don't invent XML languages"<sup>3</sup>.

Considering what we have said before, an XML application or XML vocabulary consists of more than a bunch of similar XML documents. We need one or more schemas to be able to constrain XML documents to our specific structure and to be able to tell if an XML document coming into our system is a good one or not. Think about an XML document describing books where we have a `partition` element. It is meaningless. Thus there is a need for constraints and validation.

But why would we want to have several schemas ? Because schema languages don't have the same power and easiness. We might want to use a few to define different schemas for the same XML vocabularies and test XML documents in a validation pipeline.

We also need documentation to be able to understand what our XML vocabulary describes and be able to use it.

And we need a namespace, just like namespaces in C++ or packages in Java, to avoid collisions between elements and attributes.

### meaning with tags

Like in the database world:

- A relational schema gives structure to data using attributes (name and type). Think about the columns of a table. Just like an XML schema
- A relation is data at any given point in time just like an XML document
- The data conforms to the schema

<sup>1</sup>[http://en.wikipedia.org/wiki/List\\_of\\_XML\\_markup\\_languages](http://en.wikipedia.org/wiki/List_of_XML_markup_languages)

<sup>2</sup><http://xml.coverpages.org/xmlApplications.html>

<sup>3</sup><http://www.tbray.org/ongoing/When/200x/2006/01/08/No-New-XML-Languages>

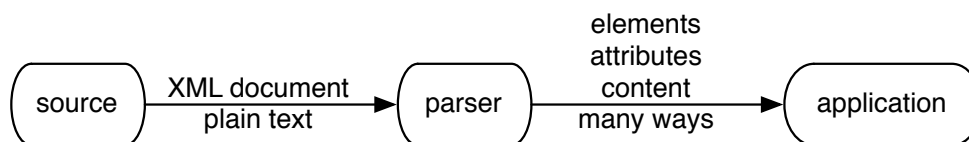
### 1.1.2 XML ecosystem

#### power

- Given that freedom, XML is powerful to describe a lot of things
- But it is quite useless without partner technologies like schemas, namespaces, XPath, XQuery, XSLT, XSL-FO, XML Pipeline, XInclude, XPointer, XLink, XForms, ...
- You can get them on the W3C Web site at <http://www.w3.org/standards/xml/>
- Not everything related to XML is made by the W3C. There are more elsewhere like Relax NG, Schematron, ...

### 1.1.3 Data flow

#### data flow



- The **source** provides an XML document. It can be a file, a database, the network, ...
- The **parser** reads the document to extract informations (structure and content)
- The software **application** gets the data in a useable form

We are interested in using programs with XML technologies. Thus the first thing we want to do is read an XML document or write one, and we want to do it often. That explains why this picture is so important.

The parser reads bytes which form characters from the source and does all the low level work to show the XML document in a much more useable form to your application program, using higher components than bytes and characters. In fact the application program gets something made of elements, attributes and other XML stuff or even objects, in the OOP sense, that have nothing to do with XML (no notion of an element or an attribute).

The application program **gets the information represented by the XML document, not the physical document itself**, that is the stream of bytes wherever it is coming from. This distinction is important and the question that arises from that is: what does the application get ? We'll see that later.

#### end-of-line handling

- In text files, the end-of-line (EOL) may be indicated by using one the following sequence: `#xA` (line feed), `#xD` (carriage return) or `#xD#xA`
- An XML parser normalizes EOL on input, before the actual parsing occurs
- EOL normalization consists of having only one kind of EOL: `#xA`
- Thus a parser only sees Unix-like style of EOL (`#xA`) when it is parsing the XML document

## 1.2 Physical and logical structures

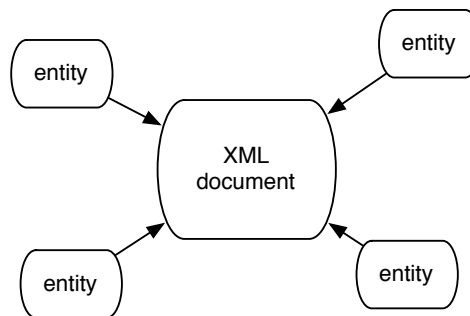
### structure

- An XML document has a **physical** and a **logical** structure
- The final document is physically assembled with entities. Think about C and `#include` directives
- The logical document is composed of various objects wherever they come from. Think about a C program once the preprocessor has processed all `#include` directives. The directives have disappeared

### 1.2.1 Physical structure

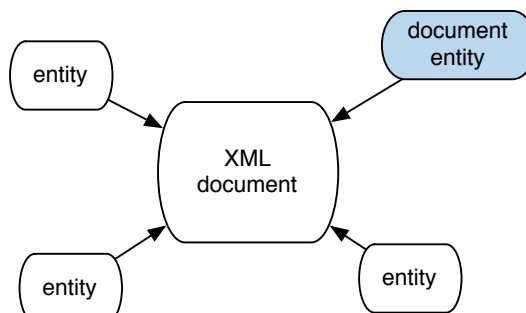
#### physical structure

- An XML document may consist of one or many storage units called **entities**
- They have content and are identified by their entity name (except for the document entity and the external DTD subset)



#### document entity

- The **document entity** serves as the root of the entity tree and a starting-point for a parser
- It has no name
- It might well appear on a processor input stream without any identification at all



**entity**

- An **entity** is a storage unit that is a piece of the XML document (or of the DTD)
- Has a name and a content called the **replacement text**
- Can be anything: a string, a file, a record in a DB, ...
- Accessed through an **entity reference**

**entity**

- Entities come in two flavors:
  - **General entities** for **use** within the document content. Must conform to the XML rules
  - **Parameter entities** for **use** within the DTD. We'll see them later
- All entities are **declared** within the DTD
- Please note the difference between utilization and declaration.

**general entities**

Four types of general entities:

- Built-in or predefined entities
- Internal text entities. We'll see them later
- External text entities. We'll see them later
- Unparsed (binary) entities. We won't see them

Unparsed entities are a way to refer to non XML stuff inside an XML document like binary data. See <http://www.w3.org/TR/xml/#dt-unparsed> if you want extra information.

**general entity reference**

- Points to an entity. It is like an alias or a pointer
- Number 6 in our example

**Syntax**

`&name_of_entity;`

**built-in entities**

Entity reference	Replacement text
<code>&amp;lt;</code>	<code>&lt;</code>
<code>&amp;gt;</code>	<code>&gt;</code>
<code>&amp;amp;</code>	<code>&amp;</code>
<code>&amp;quot;</code>	<code>"</code>
<code>&amp;apos;</code>	<code>'</code>

- Used to escape special characters
- These characters can confuse the parser if written as is

Quoted from the XML specification:

The ampersand character (&) and the left angle bracket (<) **MUST NOT** appear in their literal form, except when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. If they are needed elsewhere, they **MUST** be escaped using either numeric character references or the strings &amp; and &lt; respectively.

The right angle bracket (>) may be represented using the string &gt;; and **MUST**, for compatibility, be escaped using either &gt; or a character reference when it appears in the string ]]> in content, when that string is not marking the end of a CDATA section.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup and does not include the CDATA-section-close delimiter, ]]>. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, ]]>.

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as &apos;;, and the double-quote character (") as &quot;;.

### 1.2.2 Logical structure

#### logical structure

- Once the document has been assembled, it consists of various things or objects
- It represents the information stored in the document. This is what the application program gets<sup>4</sup>
- The things in question have been defined formally in a W3C recommendation called the **Infoset**
- XML Information Set
  - 2nd edition
  - since 2004/02/04
  - <http://www.w3.org/TR/xml-infoset/>

#### Infoset

- An XML document Infoset consists of a number of **information items**: document, element, attribute, processing instruction, unexpanded entity reference, character, comment, document type declaration, unparsed entity, notation and namespace
- Each information item (II) has a set of associated named **properties**
- We can see this as a tree structure but it can be made available to an application as anything else. E.g. event-based interface like Sax or TrAX

Note that the Infoset may contain entity references which we categorized as physical objects and thus not belonging to the logical structure. But we can ignore that fact safely for a first introduction to XML. The text below is quoted from the Infoset recommendation:

A unexpanded entity reference information item serves as a placeholder by which an XML processor can indicate that it has not expanded an external parsed entity. There is such an information item for each unexpanded reference to an external general entity within the content of an element. A validating XML processor, or a non-validating processor that reads all external general entities, will never generate unexpanded entity reference information items for a valid document.

<sup>4</sup>More or less, since other data models are used by various APIs. Anyway, it is close to the Infoset so we get the right idea.

## 1.3 XML constituents

### XML constituents

- What lies in an XML document ?
  - The XML declaration
  - Processing instructions
  - Elements
  - Attributes
  - Character references
  - Entity references
  - CDATA sections
  - Comments
  - Character data
- We are going to explain them all

### first document

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE book SYSTEM "../DTD/book.dtd">
3 <?xml-stylesheet href="..." type="..."?>
4 <book added_at="2011/07/15 15:02">
5   <isbn>0-596-00197-5</isbn>
6   <title>Java & XML</title>
7   <author>
8     <firstname>Brett</firstname>
9     <lastname>McLaughlin</lastname>
10  </author>
11  <publisher>O'Reilly</publisher>
12  <edition>2nd</edition>
13 </book>
```

### document structure

- An XML document contains only text
- It is divided in two parts:
  - The header contains all the declarations (XML, DOCTYPE and PI's)
  - The body contains XML datas within elements and attributes
- Comments and PI's may appear anywhere after the XML declaration

### 1.3.1 Root element

#### root element

- One element contains all the other elements
- It is called the **document element** or **root element**
- It is book in our example

### 1.3.2 XML declaration

#### XML declaration

---

- Optional but highly desirable
- First line in the document. Nothing before, not even a space !
- Gives information to the parser via 3 attributes: `version`, `encoding` and `standalone`
- Number 1 in our example

#### XML declaration

---

- Matches the version of the XML specification used in the document
- Could be 1.0 or 1.1
- XML 1.1 is too complex for no real benefit. Thus always use 1.0

#### XML declaration

---

- Character encoding used to write the document
- By default: UTF-8 (a superset of ASCII)
- If no indication, the parser may try to guess the encoding by using the first several bytes

#### XML declaration

---

- Can be `yes` or `no` (by default)
- Indicates if the document makes use of things defined externally<sup>5</sup>
- Behavior depends on validating or not and 4 other factors. Useless complexity
- It is always ok to say `no` and it is the default
- Thus, *forget it* unless you want to check the details

### 1.3.3 Characters

#### characters basics

---

- A **character set** represents the characters available. Each character is associated with a number, its code point
- A **character encoding** specifies how to map code points to octets
- Famous encodings: ISO-8859-1 and UTF-8 for Unicode.

---

<sup>5</sup>Indicates whether the external DTD subset affects the content of the document or not

### 1.3.4 Processing instructions

#### processing instructions

---

- Syntactically: `<?target instructions ?>`
- Given as is by the parser to the application  $\Rightarrow$  used to convey informations to the application
- PI's beginning with `xml` are forbidden
- Number 3 in our example

### 1.3.5 Elements

#### elements

---

- An **element** consists of:
  - A **start-tag** which name is an XML name like "book" or "title" in our example
  - A **content** = elements and/or character data. May be empty
  - An **end-tag**

#### The element `author` in our book example

- Element: lines 7 to 10 (with Wses)
- Start-tag: line 7 (without any WS)
- End-tag: line 10 (without any WS)
- Content: everything in between start-tag and end-tag. Wses after the start-tag on line 7, line 8 and 9, Wses before the end-tag on line 10

#### empty element

---

- An element may be empty
- It is written `<name></name>` without a single character (even a whitespace) between tags
- Shortcut, less error-prone: `<name/>`
- Note that it is allowed to contain attributes. Thus even if the content is empty, information may be conveyed in attributes

#### XML name

---

- Contain alphanumeric characters from the character set, underscore `_`, dash `-` and dot `.`
- A letter or an underscore may begin a name
- Case sensitive
- The colon `:` is reserved for namespaces
- `http://www.w3.org/TR/xml/#NT-Name` in the specification



### 1.3.6 Attributes

#### attributes

- Pairs a name and a value
- Syntax of a value: <http://www.w3.org/TR/xml/#NT-AttValue>
- The attributes of an element are enclosed in the start-tag
- Order is not significant

#### Syntax

name="value" or name='value'

#### An attribute in our book example

In the document content, we have only one attribute named `added_at`.

#### attribute-value normalization

- The normalization process occurs before the attribute value is given to the application or checked for validity
- The algorithm described at <http://www.w3.org/TR/xml/#AVNormalize> consists of:
  1. We start with a normalized value consisting of the empty string
  2. For each character, entity reference, or character reference in the unnormalized value, in order, do the following:
    - Character reference: append the referenced character to the normalized value
    - Entity reference: recursively apply step 2 to the replacement text of the entity
    - White space character (`#x20`, `#xD`, `#xA`, `#x9`): append a space character (`#x20`) to the normalized value
    - Another character, append the character to the normalized value
- If the attribute type is not CDATA, then it discards any leading and trailing space (`#x20`), and replaces sequences of space by a single space

#### attribute-value normalization

- In the source XML document, `\t` is a tabulation and `\n` a line feed.
- Let's use a Java program to parse the document and print the attributes

#### Source XML document

```
<root att1="\tthis\nis  a\n\ntest  " att2="&#x9;this&#xA;is &#x20;&#x20;a\n\ntest  "/>
```

#### Output of Java program

```
⇒NAME:att1 VALUE:" this is  a test  "
⇒NAME:att2 VALUE:⇒this
⇒is  a test  "
⇒
```

### 1.3.7 Elements vs attributes

#### elements vs attributes

- When to use elements and when to use attributes ?
- There is no definitive answer but we can give advice
- See [Effective XML] item 12

#### Attributes are...

- Good to store metadata, not the data itself. Think about `class` or `id` in XHTML
- Attribute values are just strings without structure (accessible to the parser)

#### Elements are...

- Good to store structured data
- And everything else that is not metadata

### 1.3.8 Character references

#### character references

- Refers to a specific character in the ISO/IEC 10646 character set
- Also called the Universal Character Set (UCS)
- Close to Unicode
- The character code is given in decimal or hexadecimal form

#### Decimal syntax


`&#decimal;`

#### Hexadecimal syntax

`&#xhexadecimal;`

#### character references

```
<?xml version="1.0" encoding="utf-8"?>
<text>&#x0BB9;&#x06B5;</text>
```

- Those characters have the glyphs 
- Check <http://www.utf8-chartable.de/> for a table of Unicode characters with their UTF-8 encoding

The UTF-8 encoding of code point 0BB9 corresponding to character TAMIL LETTER HA is e0aeb9. For code point 06B5 corresponding to character ARABIC LETTER LAM WITH SMALL V it is dab5. You can check that by writing a file with those bytes and opening it with an UTF-8 aware text editor.

### 1.3.9 CDATA sections

#### CDATA sections

```
<![CDATA[
<p>Un extrait de code <i>HTML</i>
dans un document <b>XML</b></p>
]]>
```

instead of

```
&lt;p&gt;Un extrait de code &lt;i&gt;HTML&lt;/i&gt;
dans un document &lt;b&gt;XML&lt;/b&gt;&lt;/p&gt;
```

- Allows us to include raw text that won't be parsed as markup
- Frees us from using a lot of entity references

### 1.3.10 Comments

#### comments

```
<!--
Ceci est un
commentaire sur plusieurs
lignes.
-->
```

- Starts with a <!--
- Ends with a -->
- Characters -- are forbidden inside comments

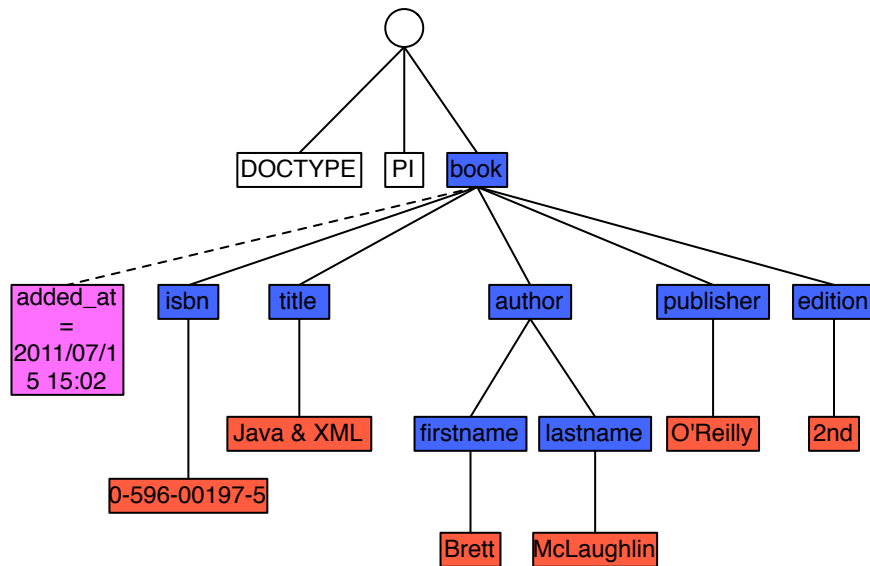
## 1.4 Tree structure

#### tree structure

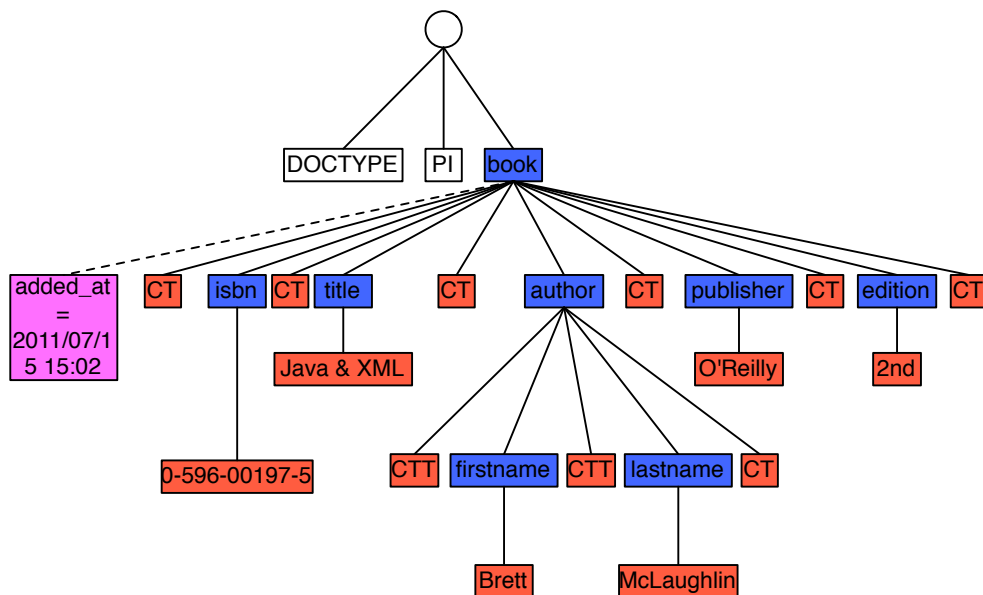
- An XML document can be represented as a tree
- Useful with DOM, XSLT, XPath, XQuery, ...
- The root of the tree does not map to anything in the serialized XML document. In particular, it is not the root element. In fact we can say that it is the document itself

The root node of the tree is abstract because we cannot see it in the file (e.g.) containing the XML document (what we called the serialized XML document). Every data structure that is a tree must have a top node called the root node of the tree. Thus we created this node to fulfill this role. It represents the whole document and is called "document" in the XML specification.

#### tree structure



### tree structure



Element nodes in blue, text nodes in red. C for Carriage Return (CR), T for TAB, CT for CR+TAB and CTT for CR+TAB+TAB. All ignorable WS nodes are present in the tree.

How can we know that some text nodes are ignorable WS ? We will have to use a schema. Cfr. next lesson.

### tree navigation

- We use the genealogical vocabulary to name things
- title is a **child** of book which is its parent
- title and authors are **siblings**
- book is the **ancestor** of every element node

- Every element node is a **descendant** of book

#### tree structure and data models

---

- Note that the tree structure presented here is quite loose. It is not a data model
- In fact, the only tool we have for now is the Infoset and it is quite abstract. It allows us to put names on the things in an XML document
- That means that every subsequent technology will have to define its own data model or reuse an existing one provided by another technology

## 1.5 Well-formedness

#### well-formedness constraint

---

- If the document conforms to some basic rules, we say that it is **well-formed** because it respects the well-formedness constraint
- It is **mandatory** for every XML document. In fact a document can only be qualified as XML if it is well-formed
- A parser is **not allowed** to read a document that is not well-formed. Thus the parser has to stop parsing when it discovers a problem

#### some basic rules

---

- Every start-tag must have a matching end-tag
- Elements may nest but not overlap. We say that they must be properly nested
- There must be exactly one root element
- Attribute values must be quoted by double or single quotes
- No two attributes with the same name in the same element
- No unescaped < or & may occur in the character data of an element or an attribute

#### checking for well-formedness

---

- A graphical tool like Oxygen<sup>6</sup>, Netbeans, Altova XMLSpy<sup>7</sup> or Stylus Studio<sup>8</sup>
- A Web site providing that kind of service like RUWF?<sup>9</sup>, RXP<sup>10</sup> or STG<sup>11</sup>
- A parser and its API like Apache Xerces Java 2, JDOM, XOM, ... We'll see some of them later

#### checking for well-formedness

---

#### Java command

---

<sup>6</sup><http://www.oxygenxml.com/>

<sup>7</sup><http://www.altova.com/>

<sup>8</sup><http://www.stylusstudio.com/>

<sup>9</sup><http://www.xml.com/pub/a/tools/ruwf/check.html>

<sup>10</sup><http://www.cogsci.ed.ac.uk/~richard/xml-check.html>

<sup>11</sup><http://www.stg.brown.edu/service/xmlvalid/>

```
$ java -cp /Users/ludo/Library/Java/xerces-2_11_0/xercesSamples.jar:/Users/ludo/
Library/Java/xerces-2_11_0/xercesImpl.jar sax.Counter src/xml/book.xml
```

Two outcomes possible:

- The file is well-formed. We just have statistical output.
- The file is not well-formed. The end-tag of the title element was written `</titlee>`. So when the parser encounters the second "e", it was expecting a ">". That explains the error message.

### checking for well-formedness

---

#### Well-formed

```
src/xml/book.xml: 46 ms (7 elems, 1 attrs, 17 spaces, 36 chars)
```

#### Not well-formed

```
[Fatal Error] book.xml:5:30: The end-tag for element type "title" must end with a
'>' delimiter.
```

## 1.6 Translations

### in french

---

English	French
entity	entité
entity reference	appel d'entité
character reference	appel de caractère
element	élément
tag	balise
start-tag	balise ouvrante
end-tag	balise fermante
well-formed	bien formé
valid	valide
well-formedness constraint	contrainte de forme
validity constraint	contrainte de validité

## Chapter 2

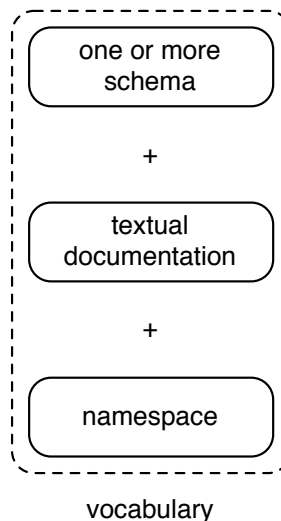
# Schema Languages and Schemas

### 2.1 Schemas

#### what are schemas ?

- A schema is a set of rules written in a specific language that one or many XML document(s) may respect
- It is used to constrain XML documents
- An XML vocabulary has one or more schema
- There are many schema languages used to write schemas

Remember the following picture:



We could say that a schema  $S$  defines a vocabulary  $V$  but it would be a little bit oversimplified since all the constraints of vocabulary  $V$  might not be expressed in the schema  $S$  and even might not be written in the given schema language. We should see that schema  $S$  and vocabulary  $V$  are associated together and that schema  $S$  is used in the process of validation to determine if a given XML document belongs to vocabulary  $V$  or "is made of" vocabulary  $V$ .

We also might want to use many schemas using different languages to validate documents against vocabulary  $V$  since different schema languages have different powers of expression.

#### DB equivalence

- A schema in the XML world is like a relational schema in the DB world created by `CREATE TABLE`
- Both of them constrain the data allowed in objects (document or table)
- Both of them define some kind of a type of objects

## 2.2 Schema languages

### schema languages

- A schema language is used to define schemas. It is the notation or syntax used to write schemas
- There are four well-known schema languages: Document Type Definition (DTD), W3C XML Schema, Relax NG and Schematron

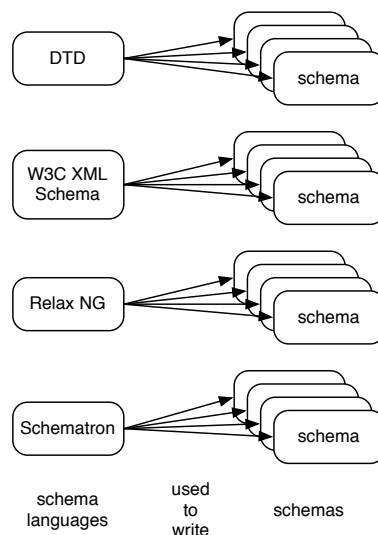
Name	Complexity	Power	Syntax
W3C DTD	simple	limited	own syntax
W3C XML Schema	complex	powerful	XML
Relax NG	simple	powerful	XML + own syntax
Schematron	complex	unlimited	XML (XSLT)

- W3C DTD: inclus dans la spécification XML à <http://www.w3.org/TR/xml/>
- W3C XML Schema: <http://www.w3.org/XML/Schema>
- Relax NG: <http://relaxng.org/>
- Schematron: <http://www.schematron.com/>

### DB equivalence

- A schema language in the XML world is like the SQL language in the DB world used to create schemas with `CREATE TABLE`
- Both of them are a syntax to communicate with the software

### graphically





**sample document**

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <addressBook>
3   <card>
4     <name>John Smith</name>
5     <email>js@example.com</email>
6   </card>
7   <card>
8     <name>Fred Bloggs</name>
9     <email>fb@example.net</email>
10  </card>
11 </addressBook>

```

**DTD**

```

1 <!ELEMENT email (#PCDATA)>
2 <!ELEMENT name (#PCDATA)>
3 <!ELEMENT card (name, email)>
4 <!ELEMENT addressBook (card*)>

```

**W3C XML Schema**

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="email" type="xs:string"/>
4   <xs:element name="name" type="xs:string"/>
5   <xs:element name="card">
6     <xs:complexType>
7       <xs:sequence>
8         <xs:element ref="name"/>
9         <xs:element ref="email"/>
10      </xs:sequence>
11    </xs:complexType>
12  </xs:element>
13  <xs:element name="addressBook">
14    <xs:complexType>
15      <xs:sequence>
16        <xs:element minOccurs="0" maxOccurs="unbounded" ref="card"/>
17      </xs:sequence>
18    </xs:complexType>
19  </xs:element>
20 </xs:schema>

```

**Relax NG full**

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <grammar xmlns="http://relaxng.org/ns/structure/1.0"
3   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
4   <start>
5     <element name="addressBook">
6       <zeroOrMore>
7         <element name="card">
8           <element name="name">
9             <text/>
10          </element>
11        <element name="email">

```

```

12         <data type="token">
13             <param name="pattern">\w+@[\w\..]+</param>
14         </data>
15     </element>
16 </element>
17 </zeroOrMore>
18 </element>
19 </start>
20 </grammar>

```

### Relax NG compact

```

1 element addressBook {
2     element card {
3         element name { text },
4         element email {
5             xsd:token { pattern = "\w+@[\w\..]+" }
6         }+
7     }*
8 }

```

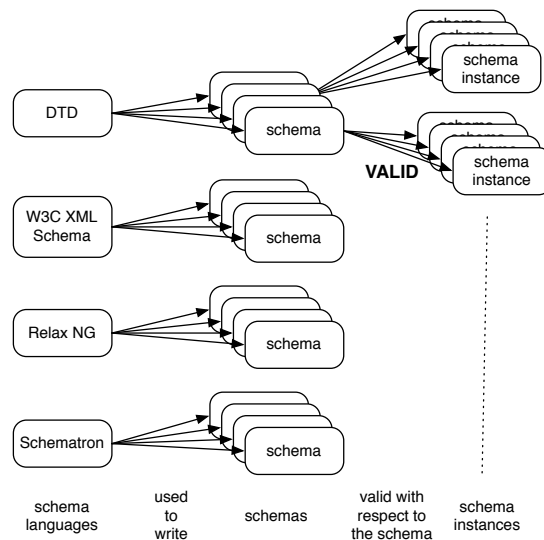
## 2.3 Schema instances and validity

### schema instances

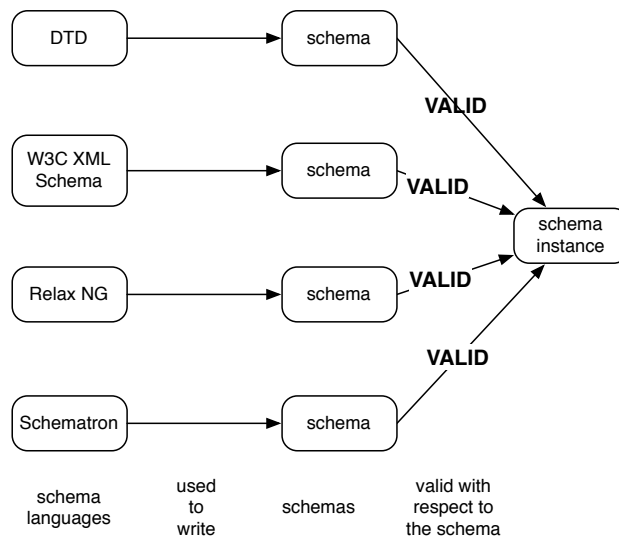
- An XML document may be an instance of a schema
- When it is, it conforms to the schema and is **valid** in respect to that schema
- In the DB world, a relation is an instance of a schema when it conforms to its constraints. In fact, data in a table always conforms to the schema if constraints are always enforced and thus is always an instance of the schema
- Note that an XML document may be the instance of multiple schemas if we need the power of different schema languages to express our constraints.

Validity is an important notion in the XML world.

### graphically

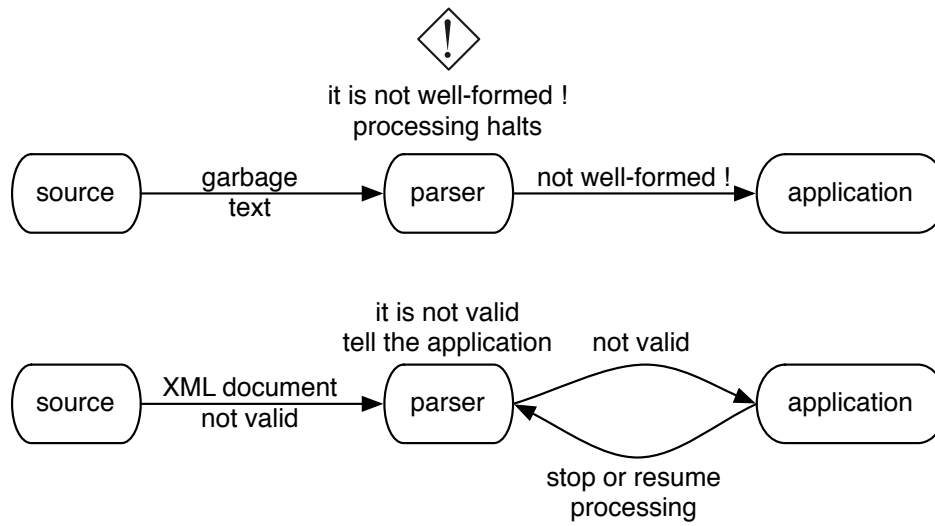


### one instance multiple schemas



### validity is optional

- Well-formedness is required by the parser
- Validity is required by an application but not by the parser. It is optional
- Validity is checked by the parser on behalf of the application



## Chapter 3

# Document Type Definition (DTD)

### 3.1 Presentation

#### definition

---

- DTD = Document Type Definition
- DTD is part of the XML recommendation
- It was the first schema language available
- No XML syntax, easy to use but often too simple
- It is used to:
  - Define constraints on elements: name and content
  - Define constraints on attributes: name, content and type
  - Define new entities

#### limited power

---

- Constraints are coarse
- No types in the usual sense: integers, values less than or greater than, textual patterns, enumeration, length, ...
- Except for some specific cases with attributes

### 3.2 Declarations

#### document type declaration

---

- The document type declaration allows us to link an XML document to its DTD
- It is written in the XML document with a DOCTYPE declaration
- There are four types of declaration:
  - Internal declaration
  - External system declaration
  - External public declaration

- External + internal declaration
- Same as "declaration vs definition" in the C programming language

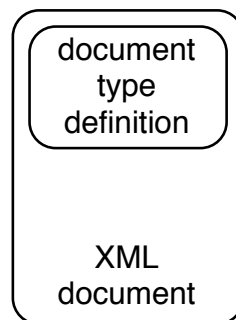
### 3.2.1 Internal declaration

#### internal declaration

```
<!DOCTYPE root element [ ... ]>
```

- The DTD is fully contained in the XML document
- Not really useful because normally one DTD maps to a class of XML documents and not just a single one. You don't want to copy/paste the DTD in each of your XML documents
- Declaration and definition are merged together
- However it is useful when you want to have self contained XML documents: content and schema together in one file to make tests

#### graphically



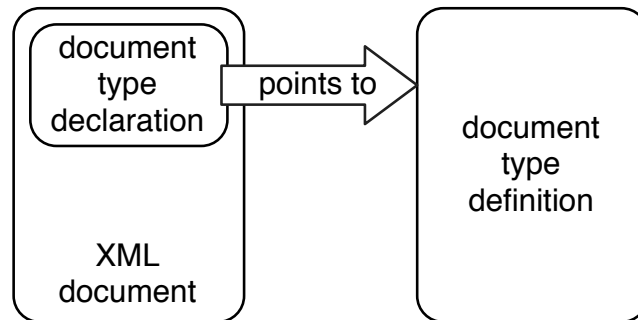
### 3.2.2 External system declaration

#### external system declaration

```
<!DOCTYPE root element SYSTEM "URI">
```

- The DTD is completely outside of the XML document
- Typical use of DTDs
- Declaration and definition are separate

#### graphically



### 3.2.3 External public declaration

#### external public declaration

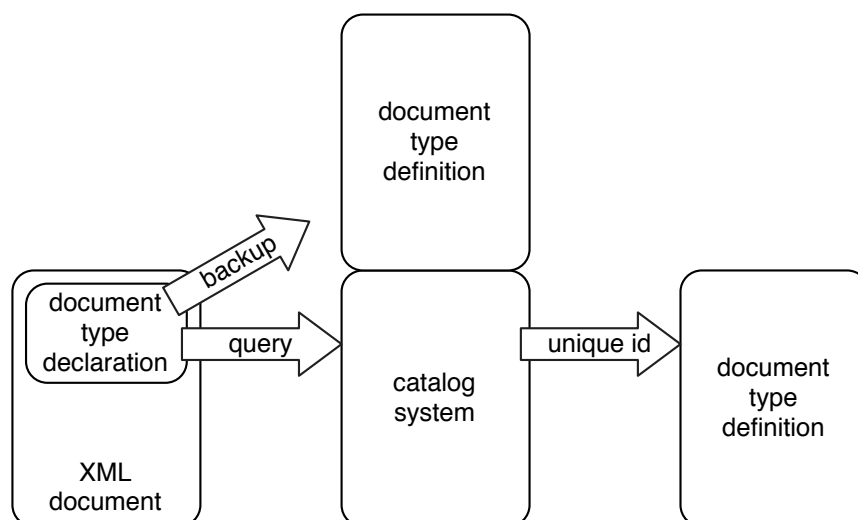
```
<!DOCTYPE root_element PUBLIC "Unique id" "Backup URI">
```

- The DTD is completely outside of the XML document
- A system of catalog helps us locate the DTD. Like the DOCTYPE of XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

- Backup URI in case of catalog unavailability
- Declaration and definition are separate

#### graphically



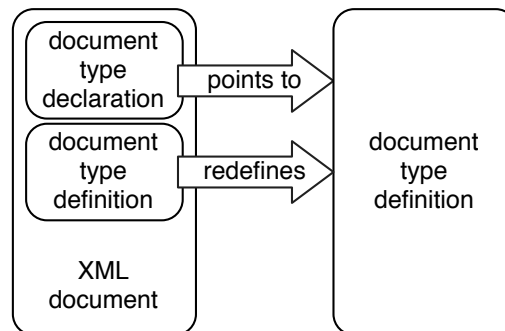
### 3.2.4 External and internal declarations

#### external + internal declaration

```
<!DOCTYPE root_element SYSTEM "URI" [ ... ]>
<!DOCTYPE root_element PUBLIC "Unique ID" SYSTEM "Backup URI" [ ... ]>
```

- The DTD is mostly outside of the XML document but also inside of it
- It allows us to declare new elements, attributes and entities, and to redefine entities (customization) in the internal DTD subset

graphically



### 3.3 Catalogs

what is a catalog ?

- A document describing a mapping between external entity references and locally-cached equivalents
- A resolver looks for the resources where the catalog points to them
- Advantages:
  - It makes us independent of the true location of the resources in our XML documents
  - It makes us independent of network downtime
- The process is called *entity resolution* and is detailed in the document "XML Catalogs" accessible at <http://www.oasis-open.org/committees/entity/>
- Editors and program code can benefit from it

catalog example

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
3   <public publicId="-//W3C//DTD XHTML 1.0 Strict//EN" uri="file:///Users/ludo/
4     Documents/cours/xml/dtd/xhtml11-strict.dtd"/>
5   <system systemId="http://cours.khi.be/dtd/library.dtd" uri="file:///Users/ludo/
6     Documents/cours/xml/dtd/library.dtd"/>
7   <public publicId="-//OASIS//DTD DocBook V5.0//EN" uri="file:///Users/ludo/
8     Documents/cours/xml/dtd/docbook.dtd"/>
9 </catalog>
```



**catalog example**

Note that in the previous example:

- The public ID  `-//W3C//DTD XHTML 1.0 Strict//EN` is mapped to a local file named `xhtml1-strict.dtd`
- The system ID `http://cours.khi.be/dtd/library.dtd` is mapped to a local file and thus no network access will occur
- The public ID  `-//OASIS//DTD DocBook V5.0//EN` is mapped to a local file

In Oxygen, if we use the catalog, we can see that:

- The catalog is used when we refer to the SYSTEM URI `http://cours.khi.be/dtd/library.dtd` in `library.xml`
- The catalog is used when we refer to the PUBLIC identifier  `-//W3C//DTD XHTML 1.0 Strict//EN` in `test.html`

Listing 3.1: File library.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE library SYSTEM "http://cours.khi.be/dtd/library.dtd">
3 <library>
4   <book isbn="0-596-00197-5">
5     <title>Java & XML</title>
6     <author>
7       <firstname>Brett</firstname>
8       <lastname>McLaughlin</lastname>
9     </author>
10    <publisher>O'Reilly</publisher>
11    <edition>2nd</edition>
12  </book>
13  <book isbn="0-262-06217-8">
14    <title>Essentials of Programming Languages</title>
15    <author>
16      <firstname>Daniel</firstname>
17      <lastname>P. Friedman</lastname>
18    </author>
19    <author>
20      <firstname>Mitchell</firstname>
21      <lastname>Wand</lastname>
22    </author>
23    <author>
24      <firstname>Christopher</firstname>
25      <lastname>T. Haynes</lastname>
26    </author>
27    <edition>2nd</edition>
28  </book>
29 </library>

```

Listing 3.2: File test.html

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
3 xhtml1/DTD/xhtml1-strict.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-BE" lang="fr-BE">
5   <head>
6     <title>Le langage HTML</title>
7     <link rel="stylesheet" type="text/css" href="labo2.css" media="screen"/>
8   </head>
9   <body>
10    <h1>La structure du document</h1>
11    <h2>L'en-tête</h2>

```

```

12      <p>L'en-tête d'un document HTML est contenue dans l'élément <code>head</code>
      c   entre les balises (tags) <code>&lt;head&gt;</code> et <code>&lt;/head&gt;
      </code>. Celle-ci contient des informations sur le document comme par exemple
      son titre.</p>
13      <p>Le titre est indiqu   dans l'en-t  te du document entre les balises <code>&
      lt;title&gt;</code> et <code>&lt;/title&gt;</code>.</p>
14
15      <h2>Le corps</h2>
16      <p>Le corps du document HTML contient les informations qui seront affich  es
      dans le navigateur et se trouve dans l'  l  ment <code>body</code>. Il s'agit de
      la plus grosse partie du document.</p>
17
18      <h1>Les titres de section</h1>
19      <p>Le document HTML peut   tre divis   en sections, sous-sections, etc...    l'
      instar de ce qui se fait avec un livre. Plusieurs niveaux de titre sont
      disponibles (6) indiqu  s    l'aide des   l  ments <code>h1</code>, <code>h2</code>
      >, <code>h3</code>, <code>h4</code>, <code>h5</code> et <code>h6</code>.</p>
20
21      <h1>Les structures de bloc (DIV)</h1>
22      <p>Les   l  ments <code>div</code> permettent de structurer le code HTML en blocs.
      Il s'agit simplement de d  limiter des parties du document. Ceci sera tr  s utile
      lorsqu'il s'agira de mettre en forme visuellement le document    l'aide des
      feuilles de style (CSS).</p>
23      <div>
24          <p>Ce texte est contenu dans un bloc <code>div</code>.</p>
25      </div>
26
27      <h1>Les listes</h1>
28      <p>Il existe trois types de liste dont les listes non num  rot  es (  l  ment <code>ul
      </code>, unordered) et les listes num  rot  es (  l  ment <code>ol</code>, ordered).
      Chaque entr  e dans la liste est indiqu  e    l'aide de l'  l  ment <code>li</code>.</p>
      >
29      <p>Par exemple, une liste non num  rot  e</p>
30      <ul>
31          <li>un</li>
32          <li>deux</li>
33      </ul>
34      <p>et une liste num  rot  e</p>
35      <ol>
36          <li>un</li>
37          <li>deux</li>
38      </ol>
39      </body>
40  </html>

```

### Oxygen output

- Oxygen resolves the system identifier to its local version
- Oxygen resolves the XHTML public identifier to its local version

```

1  Severity: info
2  Description: Public: null System: http://cours.khi.be/dtd/library.dtd = file:/Users/
   ludo/Documents/cours/xml/dtd/library.dtd
3  Severity: info
4  Description: Resolved system: http://cours.khi.be/dtd/library.dtd file:/Users/ludo/
   Documents/cours/xml/dtd/library.dtd
5
6  Severity: info
7  Description: Public: -//W3C//DTD XHTML 1.0 Strict//EN System: http://www.w3.org/TR/
   xhtml1/DTD/xhtml1-strict.dtd = file:/Applications/oxygen/frameworks/xhtml/dtd/xhtml1
   -strict.dtd

```

```
8 | Severity: info
9 | Description: Resolved system: http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd file
   | :/Applications/oxygen/frameworks/xhtml1/dtd/xhtml1-strict.dtd
```

## 3.4 Syntax

### syntax

- DTD is not XML, it has its own syntax
- It is described in the XML specification
- Like in section 3.2 for elements where we find a production called `elementdecl` giving the syntax with something like BNF
- BNF is Backus-Naur Form used to describe the syntax of programming languages
- Elements at <http://www.w3.org/TR/xml/#elemdecls>
- BNF at [http://en.wikipedia.org/wiki/Backus%E2%80%93Naur\\_Form](http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)

### 3.4.1 Elements

#### elements

Any element in the XML document must be declared once in the DTD with:

```
<!ELEMENT element name element content>
```

- `element name` may be any legal XML name
- `element content` defines what is the content of the element (between start-tag and end-tag):
  - Other elements called children
  - Unstructured text called *character data* without any markup
  - A mix of both elements and text
  - Nothing. The element is empty
  - Anything. The content is not constrained at all

#### Element content: text

#### element content: text

```
<!ELEMENT name (#PCDATA)>
```

- We write `#PCDATA` between round brackets to specify that the content contains only text, no child elements
- `PCDATA` means *Parsed Character Data*. The term comes from SGML
- Data is parsed and may contain entity references
- We have no way to constrain the text itself like we do with a datatype. It's just a string of characters
- Whitespaces belong to the text
- Note that in the recommendation, this is just a specific case of mixed content

## Element content: mixed

### element content: mixed

```
<!ELEMENT name (#PCDATA | a | b | c)*>
```

- Indicates that the element contains a mix of text and child elements a, b and c like in  
`<name>this <a>is</a> a <b>short <c>sample</c></b>.</name>`
- #PCDATA has to come first in the sequence between round brackets
- It is useful because XML documents are used in two ways:
  - To store data which is processed by a program. Mixed content is rare in this case
  - To store text which is read by someone. Just like it is done with XHTML. This case corresponds mainly to mixed content

## Element content: empty

### element content: empty

```
<!ELEMENT name EMPTY>
```

- The element has no content, it is empty but it may have attributes. We are just concerned with element content here
- Even without any attribute, the element alone carries meaning. Like `<br />` in XHTML.
- In the XML document, we write `<name></name>` or `<name/>`. There isn't any single white-space in between

## Element content: any

### element content: any

```
<!ELEMENT name ANY>
```

- The element can contain child elements and/or character data, that means anything
- The only constraint is that the child elements must have been declared
- Thanks to that we can extend the document with content not specified by the schema
- Don't use it unless you know what you do

## Element content: children

### element content: children

- Mostly used in data oriented XML documents
- We have a few ways to constrain element structure in this case. It is described at <http://www.w3.org/TR/xml/#NT-children>

- A sequence of one or more child elements. The rounded brackets are mandatory

```
<!ELEMENT name (child1, ..., childn)>
```

- A choice between child elements. The rounded brackets are mandatory

```
<!ELEMENT name (child1 | ... | childn)
```

- A repetition (zero or one, zero or more, one or more) of a sequence, a choice or an element name. We cover this on the next slide

#### element content: children

- We can repeat a name, a sequence or a choice by using a suffix
- Zero or one (optionality) with the question mark ?
- Zero or more with the star \*
- One or more with the plus sign +

#### Definition of a person name

```
<!ELEMENT name ( last_name
                  | (first_name, ( (middle_name+, last_name)
                                | last_name?
                              )
                  )
                )>
```

#### remark about child elements

#### It is ok to write this

```
<!ELEMENT collection (book | movie | cd)*>
<!ELEMENT book (...)>
<!ELEMENT movie (...)>
<!ELEMENT cd (...)>
```

#### There is an unneeded extra level of hierarchy: media

```
<!ELEMENT collection (media*)>
<!ELEMENT media (book | movie | cd)>
<!ELEMENT book (...)>
<!ELEMENT movie (...)>
<!ELEMENT cd (...)>
```

### 3.4.2 Attributes

#### attributes

We declare the attributes of an element with an attribute list.

```
<!ATTLIST element_name attribute_definition*>
```

- `element_name` may be any legal XML name and refers to an element, declared or not
- `attribute_definition` defines the name, type and default of an attribute. There can be many of this. We describe them in the next slide

- If there are more than one ATTLIST for an element, they are merged
- If there are more than one attribute definition for an attribute, only the first is taken into account

#### attribute definition

name type default declaration

- The name is an XML name
- The type belongs to one of three kinds of type and is chosen among ten types :
  - The string type CDATA
  - A set of seven tokenized types
  - Two enumerated types: notations and enumerations. We won't see notations
- The default declaration tells us if the attribute is required, optional, has a default value or a fixed value
- Cfr. <http://www.w3.org/TR/xml/#NT-AttlistDecl>

### Attribute type: CDATA

#### attribute type: CDATA

- Values of the CDATA type are strings of characters
- Some characters must be escaped: <, " or ', &.
- Note that given attribute-value normalization, the string will be transformed

**Declaration of attribute added\_at of element book**

```
<!ATTLIST book added_at CDATA #IMPLIED>
```

### Attribute type: tokenized types

#### attribute type: tokenized types

- There are seven tokenized types: ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS
- ENTITY and ENTITIES are used for unparsed entities. We won't see them in this course
- ID, IDREF and IDREFS are used when we need unique IDs
- NMTOKEN and NMTOKENS are used to constrain attributes values more than CDATA

#### attribute type: ID, IDREF and IDREFS

- Values of type ID must be a XML name and identify **uniquely** the element they belong to. Unicity is guaranteed inside the XML document
- Values of type IDREF and IDREFS refers to existing ID values
- Useful to identify and refer to elements. Extensively used in XHTML

**attribute type: ID, IDREF and IDREFS (example 1)****In the XML document**

```
<chapter name="chapter.introduction">Introduction</chapter>
...
<p>See chapter <ref name="chapter.introduction"/> ...</p>
```

**In the DTD**

```
<!ATTLIST chapter name ID #REQUIRED>
<!ATTLIST ref name IDREF #REQUIRED>
```

**attribute type: ID, IDREF and IDREFS (example 2)****In the XML document**

```
<entity name="client">...</entity>
<entity name="employee">...</entity>
<generalization name="person" entities="client employee">...</generalization>
```

**In the DTD**

```
<!ATTLIST entity name ID #REQUIRED>
<!ATTLIST generalization name ID #REQUIRED entities IDREFS #REQUIRED>
```

Note that using IDs to name entities might not be a clever idea when we think about the fact that they have to be unique on the whole document. That means other things using IDs like relationships can't have the same name. They share the same flat "namespace". We might prefix their name with `entity.` to avoid collision.

That fact is even more constraining with attributes. We might imagine prefixing attribute names with the name of the entity, which itself should be prefixed. That becomes overkill.

All of this makes us think that it could be better to avoid IDs and just use the names we want by declaring them as CDATA.

**attribute type: NMTOKEN and NMTOKENS**

- An NMTOKEN is like an XML name where the first character might be any character like the others
- NMTOKENS is a list of NMTOKEN separated by spaces. See <http://www.w3.org/TR/xml/#NT-Nmtoken>
- Useful when the attribute is not allowed to contain whitespaces in the middle of the text
- Attribute-value normalization occurs before validation

**attribute type: NMTOKEN and NMTOKENS (example)****DTD**

```
<!ATTLIST test test1 NMTOKEN #REQUIRED test2 CDATA #REQUIRED>
```

**XML document is OK**

```
<test test1=" name " test2=" na me ">/>
```

**XML document is not OK: the first attribute value contains a whitespace**

```
<test test1=" na me " test2=" na me "/>
```

#### attribute type: NMTOKEN and NMTOKENS (example)

### Processing of file with SAX in Java

```
$ java -classpath ../xml AttrValueNorm nmtoken.xml
NAME:test1 VALUE:"name"
NAME:test2 VALUE:" na me "
```

### Attribute type: enumeration

#### attribute type: enumeration

- An enumeration type is a list of possible NMTOKEN values separated by | and enclosed between round brackets
- Whitespaces may be inserted anywhere
- Useful when there is a small number of items between tokens

#### Enumeration example: week days

```
<!ATTLIST schedule day ( monday | tuesday | wednesday | thursday | friday | saturday
| sunday ) #REQUIRED>
```

### Attribute default declaration

#### attribute default declaration

The default declaration gives extra information about an attribute value. There are four possibilities:

- Mandatory  $\Rightarrow$  #REQUIRED
- Optional  $\Rightarrow$  #IMPLIED
- We declare the default value as a usual attribute value between quotes
- If we add #FIXED before the default value, its value can't change. Useful with namespaces for example

#### attribute default declaration (example)

### Attributes declaration in the DTD

```
<!ATTLIST book added_at CDATA #IMPLIED
              added_in ( library | archives ) #REQUIRED
              added_by CDATA "Ludovic"
              xmlns CDATA #FIXED "http://ns.khi.be/library">
```

### Attributes in the XML document

```
<book added_at="2011/07/15 15:02" added_in="library">
...
</book>
```



**attribute default declaration (example)****Attributes declaration: parsing with Java**

```
$ java -classpath ../xml AttrValueNorm attr_default_declaration.xml
NAME:added_at VALUE:"2011/07/15"
NAME:added_in VALUE:"archives"
NAME:added_by VALUE:"Someone"
NAME:xmlns VALUE:"http://ns.khi.be/library"
NAME:added_at VALUE:"2011/07/15"
NAME:added_in VALUE:"library"
NAME:added_by VALUE:"Ludovic"
NAME:xmlns VALUE:"http://ns.khi.be/library"
```

**3.5 Example: book document****book document**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE book SYSTEM "book.dtd">
3 <book added_at="2011/07/15 15:02">
4   <isbn>0-596-00197-5</isbn>
5   <title>Java & XML</title>
6   <title>Java & XML</title>
7   <author>
8     <firstname>Brett</firstname>
9     <lastname>McLaughlin</lastname>
10  </author>
11  <publisher>O'Reilly</publisher>
12  <edition>2nd</edition>
13 </book>
```

**DTD of book document**

```
1 <!ELEMENT isbn (#PCDATA)>
2 <!ELEMENT title (#PCDATA)>
3 <!ELEMENT firstname (#PCDATA)>
4 <!ELEMENT lastname (#PCDATA)>
5 <!ELEMENT publisher (#PCDATA)>
6 <!ELEMENT edition (#PCDATA)>
7 <!ELEMENT author (firstname, lastname)>
8 <!ELEMENT book (isbn, title, author+, publisher?, edition)>
9 <!ATTLIST book added_at CDATA #REQUIRED>
```

**3.6 Entities****entities**

- There are two types of entities:
  - General entities **for use** in the XML document
  - Parameter entities **for use** in the DTD

- Every entity is **declared** in the DTD
- An entity is data stored somewhere and we use an **entity reference** to point to that data and tell the parser to retrieve it for us

#### A general entity reference

&name;

### 3.6.1 General entities

#### internal text entities

- Useful when the replacement text is used in many documents or in a few places in the same document. It gives us a single place to modify the text
- They are declared in the DTD and the text is also contained in the DTD
- Think about a copyright text

#### An internal text entity declaration in the DTD

```
<!ENTITY name "replacement text">
```

#### internal text entities (example)

#### Declaration of copyright entity

```
<!ENTITY copyright "Copyright &#x00a9; 2011 HEPL. All Rights Reserved.">
```

#### Use of copyright entity in XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE text SYSTEM "text_internal.dtd">
<text>
  <p>blah blah blah</p>
  <p>&copyright;</p>
</text>
```

#### external text entities

- Like internal text entities but the replacement text is accessible at a URI
- It should include a text declaration that looks like an XML declaration but is not part of the replacement text
- They are declared in the DTD and the text is contained elsewhere
- Think about a copyright text

#### Both forms of external text entity declaration

```
<!ENTITY name SYSTEM "URI">
<!ENTITY name PUBLIC "Unique ID" "Backup URI">
```

#### external text entities (example)

#### Declaration of copyright entity in DTD

```
<!ENTITY copyright SYSTEM "http://khi.be/entities/copyright.xml">
```

### Use of copyright entity in XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE text SYSTEM "text_external.dtd">
<text>
  <p>blah blah blah</p>
  <p>&copyright;</p>
</text>
```

#### external text entities (example)

### Text of the entity copyright.xml

```
<?xml version="1.0" encoding="UTF-8"?>Copyright &#x00a9; 2011 HEPL. All Rights
Reserved.
```

### Parsing in Java

```
$ java Dump text_external.xml
blah blah blah
Copyright © 2011 HEPL. All Rights Reserved.
```

#### validating vs non-validating parsers

- A validating parser is required to check the entire DTD and all the external parsed entities.
- A non-validating parser is required to check the document entity and the internal DTD subset. But the recommendation lets the implementation choose what to do with the external DTD subset
- We have to be particularly careful with the behavior of the parser

## 3.6.2 Parameter entities

#### parameter entities

- Parameter entities are used in the DTD, not in the XML document, unlike general entities
- They are declared in the DTD like general entities
- They are used to:
  - Introduce modularity in the DTD and structure code
  - Avoid duplication of text in the DTD
  - Change constraints of an external DTD subset by redefining entities in the internal DTD subset
- A well known example is the DTD for XHTML 1.0 Strict:
  - Readable in HTML with hyperlinks
  - Readable as text

DTD for XHTML 1.0 Strict:

- As HTML: [http://www.w3.org/TR/xhtml1/dtds.html#a\\_dtd\\_XHTML-1.0-Strict](http://www.w3.org/TR/xhtml1/dtds.html#a_dtd_XHTML-1.0-Strict)

- As text: <http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>

#### internal parameter entities

- The replacement text is in the DTD
- We use an extra % to distinguish them from general entities

#### An internal parameter entity declaration in the DTD

```
<!ENTITY % name "replacement text">
```

#### internal parameter entities (example)

A real estate agency offers studios and apartments for rent and houses for sale. For each one, we have:

- An address
- The number of rooms
- The number of bathrooms
- The price for rent or for sale

#### DTD sample without parameter entities

```
<!ELEMENT studio (address, rooms, bathrooms, rent)>
<!ELEMENT apartment (address, rooms, bathrooms, rent)>
<!ELEMENT house (address, rooms, bathrooms, sale, garden)>
```

#### internal parameter entities (example)

Since the content of elements is quite the same, we have duplication of text. We can avoid that by using internal parameter entities.

#### DTD sample with parameter entities

```
<!ENTITY % rent_price "rent">
<!ENTITY % sale_price "sale">
<!ENTITY % studio_price "%rent_price;">
<!ENTITY % apartment_price "%rent_price;">
<!ENTITY % house_price "%sale_price;">
<!ELEMENT studio (%common_content;, %studio_price;)>
<!ELEMENT apartment (%common_content;, %apartment_price;)>
<!ELEMENT house (%common_content;, %house_price;, garden %extra_house;)>
```

#### external parameter entities

- The replacement text is outside the DTD available at a URI
- A well-known example is the definition of general entities for a lot of letters and symbols in XHTML.

#### Both form of external parameter entity declaration

```
<!ENTITY % name SYSTEM "URI">
<!ENTITY % name PUBLIC "Unique ID" "Backup URI">
```

### Inclusion of symbols in the DTD of XHTML Strict

```
<!ENTITY % HTMLsymbol PUBLIC "-//W3C//ENTITIES Symbols for XHTML//EN" "xhtml
-symbol.ent">
%HTMLsymbol;
```

## 3.7 Validity

### validity constraint

- If the document conforms to the schema, we say that it is **valid** in respect to that schema because it respects the validity constraints of the schema
- For the parser, it is **optional**. For the application program, it might be mandatory. The application program has to tell the parser to check validity or not
- Note that validity implies well-formedness

### checking for validity

- A graphical tool like Oxygen<sup>1</sup>, Netbeans, Altova XMLSpy<sup>2</sup> or Stylus Studio<sup>3</sup>
- A Web site providing that kind of service like RXP<sup>4</sup> or STG<sup>5</sup>
- A parser and its API like Apache Xerces Java 2, JDOM, XOM, ... We'll see some of them later

### checking for validity

- Note the -v flag passed to sax.Counter.
- It asks for DTD validation. We would add a -s for W3C XML Schema validation

### Java command

```
$ java -cp /Users/ludo/Library/Java/xerces-2_11_0/xercesSamples.jar:/Users/ludo/
Library/Java/xerces-2_11_0/xercesImpl.jar sax.Counter -v src/dtd/book.xml
```

### checking for validity

### Valid

```
src/dtd/book.xml: 22 ms (8 elems, 1 attrs, 19 spaces, 49 chars)
```

### Not valid: there are two titles

```
[Error] book.xml:14:8: The content of element type "book" must match "(isbn,title,
author+,publisher?,edition)".
src/dtd/book.xml: 49 ms (9 elems, 1 attrs, 21 spaces, 59 chars)
```

<sup>1</sup><http://www.oxygenxml.com/>

<sup>2</sup><http://www.altova.com/>

<sup>3</sup><http://www.stylusstudio.com/>

<sup>4</sup><http://www.cogsci.ed.ac.uk/~richard/xml-check.html>

<sup>5</sup><http://www.stg.brown.edu/service/xmlvalid/>

## 3.8 Translations

in french

---

English	French
general entity	entité générale
parameter entity	entité paramètre
public identifier	identificateur publique
system identifier	identificateur système

## Chapter 4

# Namespaces in XML

### 4.1 Presentation

#### versions

---

- Namespaces in XML is a W3C recommendation
- Version 1.0
  - 3rd edition
  - since 2009/12/08
  - <http://www.w3.org/TR/xml-names/>
- Version 1.1
  - 2nd edition
  - since 2006/08/16
  - <http://www.w3.org/TR/xml-names11/>

#### definition

---

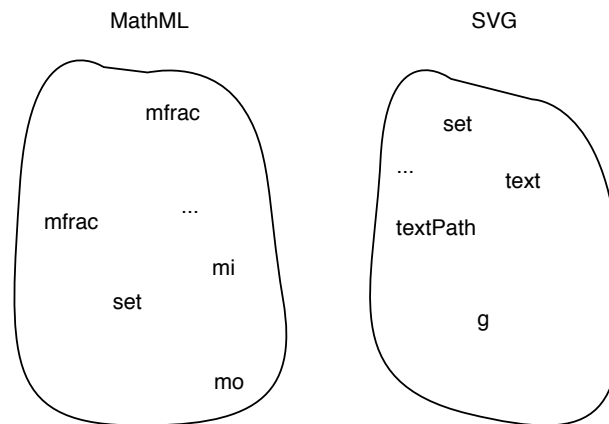
- Namespaces are a mechanism used to differentiate elements and attributes having the *same name in different vocabularies*
- We can see a namespace as a collection of elements and attributes
- It has two benefits:
  - Avoids collisions between elements and attributes with the same name that belongs to different XML vocabularies
  - Gives the ability to a program to recognize easily which vocabulary an element belongs to
- Why do that ? Because it is common to use multiple vocabularies in the same XML document. E.g. an XSLT stylesheet or a document with SVG and MathML

#### motivational example: SVG and MathML

---

- MathML and SVG both have an element named `set`
- In MathML it corresponds to mathematical sets and in SVG it allows to set a value during an animation

- They are not the same but have the same name. We distinguish them thanks to namespaces. MathML has its own namespace and SVG too.



## 4.2 Expanded names

### identification and expanded names

- An XML namespace is **identified by** a URI. The URI should be globally unique. NS URIs are compared as raw strings and it does not have to point to something accessible on the Net
- It accomplishes its goals by assigning expanded names to elements and attributes
- An **expanded name** is a pair consisting of a namespace name and a local name
- A **namespace name** is the URI of the namespace or it has no value if the element or attribute isn't in any namespace
- A **local name** is the name of the element or attribute

### NS URI examples

- Here are a few examples of namespace URIs:
  - W3C XML Schema : `http://www.w3.org/2001/XMLSchema`
  - XSLT 1.0 : `http://www.w3.org/1999/XSL/Transform`
  - XHTML 1.0 Strict : `http://www.w3.org/1999/xhtml`
  - OASIS Catalog : `urn:oasis:names:tc:entity:xmlns:xml:catalog`
- Some point to a resource, others not. It is not mandatory
- Using a URL as a URI allows people to get in control of all NS URIs starting with *their* domain name because they own it
- For example, my NS URIs could have the form `http://khi.be/ns/...`

## 4.3 Qualified names

### qualified names



- Inside the XML document, we do not use expanded names to name elements and attributes because they are inconveniently long and can contain characters not allowed in XML names
- We use qualified names which are names "subject to namespace interpretation" i.e. they may contain extra information related to namespaces
- A **qualified name** is a name which is prefixed or not
- See <http://www.w3.org/TR/REC-xml-names/#NT-QName>

```

QName      ::= PrefixedName | UnprefixedName
PrefixedName ::= Prefix ':' LocalPart
UnprefixedName ::= LocalPart
Prefix      ::= NCName
LocalPart   ::= NCName

```

### qualified names

- Thus inside the document, we use names like `prefix:name` where
  - `prefix` is called the **prefix**
  - `name` is called the **local part**
  - the whole is called the **qualified name**
- The prefix and local part must match the production `NCName` i.e. be an XML name without any colon

```
NCName ::= Name - (Char* ':' Char*)
```

## 4.4 Declaration and binding

### NS binding declaration and its scope

- We make a binding between the NS name and an arbitrary prefix
- The prefix plays the role of a short alias for the NS URI
- That binding (called a NS declaration or a NS binding declaration) has a **scope** which extends from the beginning of the start-tag in which it appears to the end of the corresponding end-tag
- We can have multiple NS declaration in an element

#### Namespace binding declaration

```

<prefix:name xmlns:prefix = "NS URI" ...>
... <!-- Scope of binding prefix <-> NS URI -->
</prefix:name>

```

### scope can be nested

An inner NS declaration redefines the existing binding while it is in scope. It is the same when a local variable shadows a global variable.

#### Inner NS declaration redefines outer NS declaration

```

<prefix:name xmlns:prefix = "NS URI 1" ...>
... <!-- Scope of binding prefix <-> NS URI 1 -->
  <prefix:name2 xmlns:prefix = "NS URI 2" ...>
    ... <!-- Scope of binding prefix <-> NS URI 2 -->
    </prefix:name2>
  ... <!-- Scope of binding prefix <-> NS URI 1 -->
</prefix:name>

```

## 4.5 XML sample document with prefixes

### XML sample document with prefixes

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE book:book SYSTEM "book_prefix.dtd">
3 <book:book xmlns:book="http://cours.khi.be/ns/book" added_at="2011/07/15
  15:02">
4   <book:isbn>0-596-00197-5</book:isbn>
5   <book:title>Java & XML</book:title>
6   <book:author>
7     <book:firstname>Brett</book:firstname>
8     <book:lastname>McLaughlin</book:lastname>
9   </book:author>
10  <book:publisher>O'Reilly</book:publisher>
11  <book:edition>2nd</book:edition>
12 </book:book>

```

### DTD of sample document with prefixes

```

1 <!ELEMENT book:isbn (#PCDATA)>
2 <!ELEMENT book:title (#PCDATA)>
3 <!ELEMENT book:firstname (#PCDATA)>
4 <!ELEMENT book:lastname (#PCDATA)>
5 <!ELEMENT book:publisher (#PCDATA)>
6 <!ELEMENT book:edition (#PCDATA)>
7 <!ELEMENT book:author (book:firstname, book:lastname)>
8 <!ELEMENT book:book (book:isbn, book:title, book:author+, book:publisher,
  book:edition)>
9 <!ATTLIST book:book
10   added_at CDATA #REQUIRED
11   xmlns:book CDATA #FIXED "http://cours.khi.be/ns/book">

```

- Note the use of a #FIXED attribute value
- Note that xmlns is part of the attribute name

### The problem with DTD and NS

- As we said, the prefix is local to the document (in fact to its scope)
- But if we put the prefix inside the DTD, we force all XML documents whose type is the DTD to use that prefix
- Thus a local prefix becomes a global one !
- Why is that ? Because DTD knows nothing about NS

- We have two solutions:
  - Use a default NS declaration but stop using prefixes
  - Use parameter entities and keep on using prefixes

## 4.6 Default namespace

### default namespace

- A default namespace declaration has the same scope of a NS declaration with a prefix
- It applies to all **unprefixed element names** within its scope
- It does not apply to attribute names
- The expanded name of such an element has the URI of the default namespace as its namespace name. No default NS declaration in scope  $\Rightarrow$  no value for the NS name
- Unprefixed attribute name  $\Rightarrow$  always no value for the NS name

### Default namespace declaration

```
<name xmlns = "NS URI" ...>
... <!-- Scope of default NS declaration -->
</name>
```

### XML sample document with default NS

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE book SYSTEM "book_default.dtd">
3 <book xmlns="http://cours.khi.be/ns/book" added_at="2011/07/15 15:02">
4   <isbn>0-596-00197-5</isbn>
5   <title>Java & XML</title>
6   <author>
7     <firstname>Brett</firstname>
8     <lastname>McLaughlin</lastname>
9   </author>
10  <publisher>O'Reilly</publisher>
11  <edition>2nd</edition>
12 </book>
```

### DTD of sample document with default NS

```
1 <!ELEMENT isbn (#PCDATA)>
2 <!ELEMENT title (#PCDATA)>
3 <!ELEMENT firstname (#PCDATA)>
4 <!ELEMENT lastname (#PCDATA)>
5 <!ELEMENT publisher (#PCDATA)>
6 <!ELEMENT edition (#PCDATA)>
7 <!ELEMENT author (firstname, lastname)>
8 <!ELEMENT book (isbn, title, author+, publisher, edition)>
9 <!ATTLIST book
10   added_at CDATA #REQUIRED
11   xmlns CDATA #FIXED "http://cours.khi.be/ns/book">
```

## 4.7 Fixing the NS problem in DTD

### fixing the NS problem in DTD

- We use parameter entities to allow the author to redefine the prefix in its XML document
- Note how the internal DTD subset is used to redefine things defined in the external DTD subset
- The prefix is named `book.prefix`
- There is a double indirection when using this technique because:

When a parameter-entity reference is recognized in the DTD and included, its replacement text must be enlarged by the attachment of one leading and one following space (`#x20`) character; (...). This behavior must not apply to parameter entity references within entity values

- See section 4.4.8 of the XML rec. 5th edition and item 7 of [Effective XML]

### XML sample document 1 with good prefix

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE book:book SYSTEM "book_ns_ok.dtd" [
3   <!ENTITY % book.prefix "book">
4 ]>
5 <book:book xmlns:book="http://cours.khi.be/ns/book" added_at="2011/07/15
  15:02">
6   <book:isbn>2011/07/15 15:02</book:isbn>
7   <book:title>Java & XML</book:title>
8   <book:author>
9     <book:firstname>Brett</book:firstname>
10    <book:lastname>McLaughlin</book:lastname>
11  </book:author>
12  <book:publisher>O'Reilly</book:publisher>
13  <book:edition>2nd</book:edition>
14 </book:book>

```

### XML sample document 2 with good prefix

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE b:book SYSTEM "book_ns_ok.dtd" [
3   <!ENTITY % book.prefix "b">
4 ]>
5 <b:book xmlns:b="http://cours.khi.be/ns/book" added_at="2011/07/15 15:02">
6   <b:isbn>0-596-00197-5</b:isbn>
7   <b:title>Java & XML</b:title>
8   <b:author>
9     <b:firstname>Brett</b:firstname>
10    <b:lastname>McLaughlin</b:lastname>
11  </b:author>
12  <b:publisher>O'Reilly</b:publisher>
13  <b:edition>2nd</b:edition>
14 </b:book>

```

**DTD with good prefix**

```
1 <!ENTITY % book.prefix "book">
2 <!ENTITY % book.colon ":">
3
4 <!ENTITY % book.qname "%book.prefix;%book.colon;book">
5 <!ENTITY % isbn.qname "%book.prefix;%book.colon;isbn">
6 <!ENTITY % title.qname "%book.prefix;%book.colon;title">
7 <!ENTITY % firstname.qname "%book.prefix;%book.colon;firstname">
8 <!ENTITY % lastname.qname "%book.prefix;%book.colon;lastname">
9 <!ENTITY % publisher.qname "%book.prefix;%book.colon;publisher">
10 <!ENTITY % edition.qname "%book.prefix;%book.colon;edition">
11 <!ENTITY % author.qname "%book.prefix;%book.colon;author">
12
13 <!ENTITY % author.content "%firstname.qname;, %lastname.qname;">
14 <!ENTITY % book.content "%isbn.qname;, %title.qname;, %author.qname;+, %
    publisher.qname;, %edition.qname;">
15
16 <!ELEMENT %title.qname; (#PCDATA)>
17 <!ELEMENT %isbn.qname; (#PCDATA)>
18 <!ELEMENT %firstname.qname; (#PCDATA)>
19 <!ELEMENT %lastname.qname; (#PCDATA)>
20 <!ELEMENT %publisher.qname; (#PCDATA)>
21 <!ELEMENT %edition.qname; (#PCDATA)>
22 <!ELEMENT %author.qname; (%author.content;)>
23 <!ELEMENT %book.qname; (%book.content;)>
24
25 <!ATTLIST %book.qname; added_at CDATA #REQUIRED>
26
27 <!ENTITY % xmlns "xmlns:%book.prefix;">
28 <!ATTLIST %book.qname; %xmlns; CDATA #FIXED "http://cours.khi.be/ns/book">
```



## Chapter 5

# APIs and Implementations

### purpose

---

- This short chapter on APIs and implementations aims to clarify things in the complex world of software implementations
- We distinguish three things:
  1. We have **specifications** of languages or technologies. E.g. XML, XPath, XSLT, XQuery
  2. We have **APIs** to be able to use the specifications with a program. E.g. parsing, building, transforming, searching
  3. Finally we have **implementations** of those APIs
- As a software developer, we choose them in order:
  1. First the technology: XML 1.0
  2. Then the API for parsing/building: Sax, DOM, JDOM, XOM, JAXB, ...? We could choose Sax
  3. Finally the implementation: commercial ? open-source ? In which language ? And then we choose the Xerces Java 2 implementation of the Apache Software Foundation

## 5.1 APIs

### APIs

---

- Each API has two main goals:
  - Provides an interface to access the XML document
  - Provides an interface to access or configure the parser
- Four main types:
  - Event-based push APIs: Sax, XNI, ...
  - Event-based pull APIs: StAX, XmlPull, ...
  - Tree-bases APIs: DOM, JDOM, XOM, ...
  - Data binding APIs: JAXB, Castor, ...
- And there are also other kind of APIs like query APIs for XSLT and XPath: TrAX which is included in JAXP

### abstraction level of APIs

---

There are two categories:

- **Low level APIs** which presents the XML document using the concepts of XML like elements, attributes, character data, processing instructions, .... Event-based and tree-based APIs fall in this category
- **High level APIs** which presents the XML document as a data source without any use of XML concepts.

#### 5.1.1 Event-based APIs

##### Event-based APIs

---

- The parser doesn't build anything in memory except a few bytes necessary to do his work. Thus the memory is independent of the size of the document. It is  $O(1)$ .
- The parsing is very fast
- But the application program has to build its own data structures to do its work. It builds and uses what it needs

##### Event-based push APIs

---

- The flow of data goes from the parser to the application. We call this **streaming**
- The **parser controls the execution** of the application program code
- The **parser pushes data** to the application
- This kind of API implements the observer design pattern like we do with GUIs. We have callback methods

##### Pseudo-code of push API

```
handler = new MyOwnHandler();
parser.setContentHandler(handler);
parser.parse();
```

##### Event-based pull APIs

---

- The flow of data goes from the parser to the application. We call this **streaming**
- The application **program controls the execution** of the parser
- The application **program pulls data** from the parser
- This kind of API implements the iterator design pattern

##### Pseudo-code of pull API

```
while (true) {
    int event = parser.next();
    if (event == END_DOCUMENT) break;
    if (event == START_ELEMENT) processElement();
}
```



### 5.1.2 Tree-based APIs

#### Tree-based APIs

- The parser builds a tree in memory and gives it to the application program
- The whole tree which represents the XML document plus extra stuff for the data structure has to reside in memory. Thus the memory usage is important and could be large. It is  $O(n)$ .
- It allows easy random access to the XML document
- This kind of structure is heavily used by implementations of XPath, XSLT, XQuery, ...

#### Event-based vs Tree-based

The pros of one kind become the cons of the other. A streaming API:

- ✓ Uses a little memory
- ✓ Is very fast
- ✓ Lets the CPU work between I/O. Do some I/O to get the next thing from the XML document, give it to the application program which does its processing, and so on...
- ✗ Lets the program build its own data structure. It could be seen as an advantage if we don't need something complex
- ✗ Doesn't allow easy access to the XML document to change or to look up something. No XPath, no XSLT

### 5.1.3 Data-binding APIs

#### Data-binding APIs

- In other APIs, classes represent the concepts of XML (elements, attributes, ...)
- In data-binding APIs, classes represent the concepts the XML document represents (persons, books, cars, ...)
- In fact, the application program doesn't even know it is using XML
- There is a mapping between a schema and the OOP classes used in the program
- Constructing OOP objects from an XML document is called **unmarshalling**
- Writing OOP objects into an XML document is called **marshalling** or **serialization**

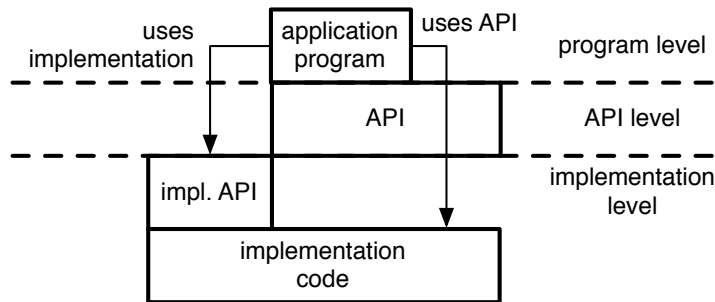
## 5.2 Implementations

#### implementations

An implementation means two things:

- A product made by an organization and implementing a few APIs like Apache Xerces Java 2 implementing Sax and DOM
- A set of classes (in a product) implementing a single API like the classes of Xerces Java 2 implementing the Sax API

### APIs and implementation

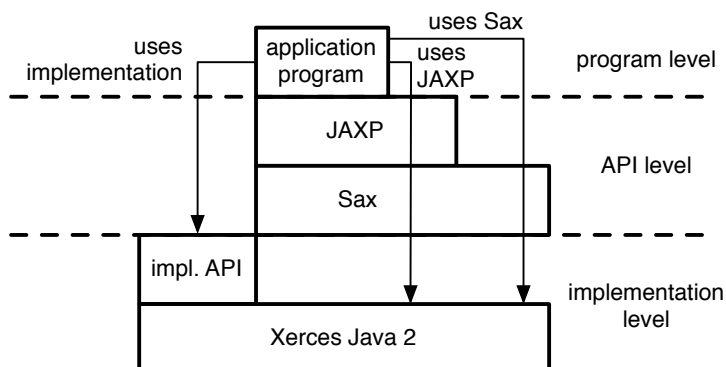


- ✓ Using a standard API is good
- ✗ Using an implementation API which is proprietary is bad because we can't switch implementations without modifying the program

### APIs and implementation (example)

- Let's take an example with JAXP, Sax and Apache Xerces Java 2.
- Sax is a standard API for push parsing
- Apache Xerces Java 2 implements the Sax API
- JAXP is a Java API that sits on top of other APIs like Sax and DOM to provide extra functionality and ease of use

### APIs and implementation (example)



## Chapter 6

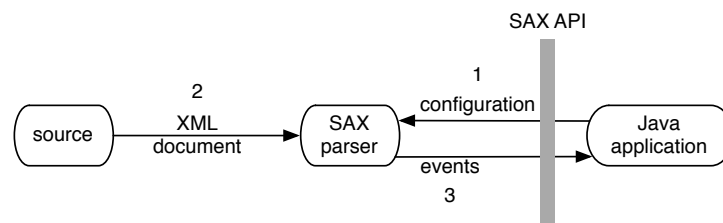
# Simple API for XML (SAX)

### 6.1 Presentation

#### presentation

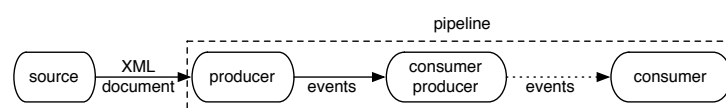
- Event-based push API
- SAX is not a standard in the usual sense but a "de facto" standard
- Originated in Java but ported to other languages
- Version 2.0.2 since 2004/04/27
- <http://www.saxproject.org/>

#### data flow



1. The application instantiates and configures the SAX parser
  2. The XML document is read from the source
  3. Events are sent to the application i.e. callback methods are called by the parser
- Step 2 and 3 occur repeatedly one after the other.

#### producer and consumer



- The parser produces events like "I have seen the start-tag of book", "I have seen character data" and send them to the application

- The application consumes the events one by one
- Note that it may form a pipeline of producers and consumers, some of the components playing both roles, like the "pipe" system in Unix

## 6.2 Java packages and documentation

### Java packages

- SAX is composed of 3 packages and it forms the Java SAX API:
  - `org.xml.sax` has the core interfaces and exceptions
  - `org.xml.sax.helpers` has utility classes implementing some core interfaces
  - `org.xml.sax.ext` has a few extensions that a parser may not implement
- We will use the Java API for XML Processing (JAXP) version 1.4 in JDK 1.6 to instantiate a SAX parser, even if SAX has its own mechanism
- It allows us to have a common interface to find implementations of SAX and DOM

### documentation

- SAX documentation is available at <http://www.saxproject.org/>. We have documentation about the API, features and properties, etc.
- JDK documentation is at <http://download.oracle.com/javase/6/docs/api/> under packages:
  - `org.xml.sax`, `org.xml.sax.helpers` and `org.xml.sax.ext` for SAX
  - `javax.xml.parsers` for SAX in JAXP

## 6.3 Example 1: parsing does nothing

### SAXExample1.java

- We configure the parser and launch the parsing but our callback methods are empty
- We actually do nothing and nothing remains after parsing

#### SAXExample1 execution on book.xml

```
$ javac SAXExample1.java
$ java SAXExample1 ../dtd/book.xml
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl
com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
Everything is ok !
```

### well-formedness and validation errors

#### book.xml is not well-formed: title end-tag is bad

```
$ java SAXExample1 ../dtd/book.xml
Parsing error: The end-tag for element type "title" must end with a '>' delimiter.
```

**book.xml is not valid: there are two titles**

```
$ java SAXExample1 ../dtd/book.xml
Everything is ok !
```

Note that validity is not mandatory for the parser. Everything is ok

**how do we parse a document ?**

In order, we do the following:

1. Locate a parser implementation and instantiate it
2. Register callback methods that will be called by the parser. Note that calling methods is the same as firing events sent to the consumer
3. Start to parse the given document

We will see those steps in details in the following slides.

**6.3.1 Parser instantiation****parser instantiation**

- How do we locate a SAX implementation i.e. a SAX parser ?
- We use the static method `newInstance` of class `SAXParserFactory`
- That method tries to locate an implementation with an ordered lookup procedure:
  1. Use the `javax.xml.parsers.SAXParserFactory` system property
  2. Use the properties file `lib/jaxp.properties`
  3. Use the Services API that will look for a classname in the file `META-INF/services/javax.xml.parsers.SAXParserFactory` in jars available to the runtime
  4. Platform default `SAXParserFactory` instance
- That means we can easily use another one by giving a value to `javax.xml.parsers.SAXParserFactory`

**parser instantiation**

1. We instantiate the factory

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

2. We check the concrete class of the factory. It is Apache Xerces

```
System.out.println(factory.getClass().getName());
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
```

3. We gives options to select/configure the parser like validation

```
factory.setValidating(true);
```

4. And eventually we instantiate the parser and check its concrete class

```
SAXParser saxParser = factory.newSAXParser();
System.out.println(saxParser.getClass().getName());
com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl
```

**from JAXP to SAX**

- We have an instance of `javax.xml.parsers.SAXParser` but we need an instance of `org.xml.sax.XMLReader` to configure the parser with our callback methods. How ?
- With the method `getXMLReader()` we get the `XMLReader` that is encapsulated by the implementation of `SAXParser`

```
XMLReader xmlReader = saxParser.getXMLReader();
System.out.println(xmlReader.getClass().getName());
com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
```

- We used JAXP to instantiate a parser and now we have full access to a SAX parser

### 6.3.2 Register callback methods

#### register callback methods

- Next, we register callback methods for different kinds of events, the most important being:
  1. Events coming from the **logical content** of the document (the infoset) with the `ContentHandler` interface
  2. Events coming from **errors** (well-formedness, validity, ...) with the `ErrorHandler` interface
- The interface `XMLReader` provides methods to register handlers for those events:
  1. `setContentHandler(ContentHandler handler)`
  2. `setErrorHandler(ErrorHandler handler)`
- Those interfaces count quite a few methods: 11 for `ContentHandler` and 3 for `ErrorHandler`. Thus, SAX provides us with a helper class `DefaultHandler` which implements all those methods with empty methods. We just have to redefine the right methods and not all of them

#### register callback methods

- We redefine nothing, thus our callback methods do nothing
 

```
DefaultHandler consumer = new DefaultHandler();
xmlReader.setContentHandler(consumer);
xmlReader.setErrorHandler(consumer);
```
- Note that `DefaultHandler` also implements two other interfaces:
  - `DTDHandler` for notations and unparsed entities
  - `EntityResolver` to allow the application to resolve entities by itself

### 6.3.3 Parsing

#### starting to parse

- Finally, we start the parsing by calling the `parse` method of `XMLReader` or `SAXParser`
  - `XMLReader`: argument is a `URI` or an `InputSource`
  - `SAXParser`: arguments are an `InputStream`, a `URI`, a `File` or an `InputSource`, and a `DefaultHandler`

```
xmlReader.parse(args[0]);
```

- An `InputSource` which encapsulates a character stream (`Reader`), a byte stream (`InputStream`), a system identifier or a public identifier
- Thus it is easy to read from a string or directly from the network

### 6.3.4 Exceptions

#### which exceptions could we get ?

- Exceptions can come:
  - **Before parsing** when we configure things
  - **While parsing**
- There are two types of exception in Java:
  - **Checked exceptions** (`Exception` and subclasses except `RuntimeException`): declared in a `throws` clause. They belong to the API and should be caught somewhere
  - **Unchecked exceptions** (`RuntimeException`, `Error`, and subclasses). Not declared because not expected. The program code has difficulties recovering from them. They don't belong to the API and we're not supposed to catch them individually
  - See Unchecked Exceptions — The Controversy and Java theory and practice: The exceptions debate

#### Exceptions: before parsing

- When we instantiate a `SAXParserFactory` we can get a `FactoryConfigurationError` error if we can't locate an implementation
- When we instantiate a `SAXParser` we can get:
  - `ParserConfigurationException`: a parser satisfying the requested configuration options cannot be instantiated
  - `SAXException`: thrown by the hidden SAX API
- When we get the `XMLReader` with `getXMLReader()` we can get a `SAXException`
- Thus we have to catch `ParserConfigurationException` and `SAXException`

#### Exceptions: while parsing

- `SAXException`: for all error conditions concerning SAX (well-formedness, validity, ...)
- `java.io.IOException`: for low-level access to the source of the XML document
- With JAXP, the runtime exception `IllegalArgumentException` might be thrown

## 6.4 Features and properties

#### features and properties : what are they ?

- A SAX parser is configured through **features** and **properties**
- A feature has a boolean value which indicates if it's activated or not

- A property has a value of type `Object`. It allows us to set things which are not just true or false
- They are identified by URIs: `http://xml.org/sax/features/` for standard SAX features and `http://xml.org/sax/properties/` for standard properties
- Only two features are required and at least we have them read-only:
  - `http://xml.org/sax/features/namespace` (true)
  - `http://xml.org/sax/features/namespace-prefixes` (false)
- Standard SAX2 features and properties, Xerces Java 2 features and Xerces Java 2 properties
- Standard SAX2 features and properties : `http://www.saxproject.org/apidoc/org/xml/sax/package-summary.html#package_description`
- Xerces Java 2 features : `http://xerces.apache.org/xerces2-j/features.html`
- Xerces Java 2 properties : `http://xerces.apache.org/xerces2-j/properties.html`

#### features and properties : how to get/set them

- In `XMLReader` interface we have methods to get/set them:
  - `boolean getFeature(String name)`
  - `Object getProperty(String name)`
  - `boolean setFeature(String name, boolean value)`
  - `Object setProperty(String name, Object value)`
- With JAXP, features are set with a `SAXParserFactory` and properties with a `SAXParser`

#### features and properties : example

- We have seen one way to ask for validation: call `setValidating(true)` on `SAXParserFactory`
- We can do the same with a feature named `http://xml.org/sax/features/validation`

#### Ask for validation with an `XMLReader`

```
// parser is of type XMLReader
parser.setFeature("http://xml.org/sax/features/validation", true);
```

## 6.5 Example 2: display data

#### SAXExample2.java

- We redefine some callback methods to display elements, attributes and character data (including ignorable whitespaces)
- We output UTF-8 and Unicode code points in hexadecimal for ignorable whitespaces

#### SAXExample2 execution on `book.xml`



```
$ javac SAXExample2.java
$ java SAXExample2 ../dtd/book.xml
startElement. uri:, localName:, qName:book
    uri:, localName:added_at, qName:added_at = '2011/07/15 15:02'
    ignorableWhitespace. [0xA, 0x9]
startElement. uri:, localName:, qName:isbn
characters. [0-596-00197-5]
endElement. uri:, localName:, qName:isbn
Characters accumulated: 0-596-00197-5
...
```

### execution without a DTD and whitespaces

- We run `SAXExample2` without providing a DTD
- Ignorable whitespaces are not recognized as such anymore. They are received through the `characters` method. Indeed, without a DTD, the parser can't tell that those whitespaces are just for indentation

### SAXExample2 execution on book.xml without a DTD

```
$ javac SAXExample2.java
$ java SAXExample2 ../dtd/book.xml
startElement. uri:, localName:, qName:book
    uri:, localName:added_at, qName:added_at = '2011/07/15 15:02'
characters. [
]
startElement. uri:, localName:, qName:isbn
...
```

### defining ContentHandler methods

- We will implement some methods of the `ContentHandler` interface to intercept start-tag, end-tag, characters and ignorable whitespaces events
- We inherit from `DefaultHandler` and redefine 4 methods:
  - `startElement` for start-tag event
  - `endElement` for end-tag event
  - `characters` for characters event
  - `ignorableWhitespace` for ignorable whitespaces event due to indentation
- We don't try to process parsing errors here. Saved for later

## 6.5.1 ContentHandler startElement

### startElement

```
public void startElement(String uri, String localName, String qName,
    Attributes atts) throws SAXException
```

- Each time a start-tag is seen by the parser, it calls our callback method `startElement`
- We get up to 3 name components for each element:
  - `uri`: the namespace URI
  - `localName`: the local name
  - `qName`: the qualified name (with prefix)

- We get the attributes through an `org.xml.sax.Attributes` object which encapsulates the set of attributes
- In case of trouble we can throw a `SAXException` and stop the parser

#### parser configuration and NS informations

Two features tell the parser how to report NS:

- `http://xml.org/sax/features/namespaces`
- `http://xml.org/sax/features/namespace-prefixes`
- `namespaces`. `true` : the NS URI and local name are required (the default). `false` : they are optional (if one is specified, both must be)
- `namespace-prefixes`. `true` : the qualified name is required. `false` : it is optional (the default)
- Set `namespaces` and `namespace-prefixes` to get all informations

#### attributes in startElement

- We can get the local name, qualified name and URI of the attribute
- We only get attributes with explicit values: specified or defaulted. Implied attributes are not reported
- We get `xmlns*` attributes only if `namespace-prefixes` is set to `true`
- See the Javadoc of `Attributes`

#### SAXExample2 execution on book.xml

```
$ java AttrValueNorm ../ns/book_ns_ok_attr.xml
http://xml.org/sax/features/namespaces : true
http://xml.org/sax/features/namespace-prefixes : true
URI: , LOCAL NAME: , QNAME:xmlns:book, VALUE:"http://cours.khi.be/ns/book"
URI:http://cours.khi.be/ns/book, LOCAL NAME:added_at, QNAME:book:added_at, VALUE:"2011/07/15
15:02"
```

Javadoc of `Attributes`: <http://www.saxproject.org/apidoc/org/xml/sax/Attributes.html>

### 6.5.2 ContentHandler endElement

#### endElement

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

- Each time a start-tag is seen by the parser, it calls our callback method `startElement`
- We get up to 3 name components for each element:
  - `uri`: the namespace URI
  - `localName`: the local name
  - `qName`: the qualified name (with prefix)
- In case of trouble we can throw a `SAXException` and stop the parser

### 6.5.3 ContentHandler characters

#### characters

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

- The parser reports a chunk of character data with this method going from `ch[start]` to `ch[start+length-1]`
- Contiguous characters may be reported by multiple calls to `characters`. But all the characters in a given call must come from the same *external* entity
- Note that each Java `char` is 16 bits wide and thus can't represent characters whose code point is  $\geq 65536$  (outside the Basic Multilingual Plane). Those characters come in surrogate pairs (high and low) and are represented as a pair of `char`. See the Javadoc of `Character`, Unicode surrogate programming with the Java language

#### Unicode planes and surrogate pairs

- All characters in Unicode are divided in 17 planes:
  - Characters with code points between 0 and FFFF go in plane 0: the Basic Multilingual Plane. They are contained in 2 bytes like the `char` in Java (UTF-16)
  - Characters (or non-characters) with code points greater than FFFF go in other planes
- Now, how do we represent characters with code points  $\geq 65536$  on 2 bytes ? We use surrogate pairs: 2 `chars` i.e. 4 bytes. A high surrogate from D800–DBFF followed by a low surrogate from DC00–DFFF
- The code point is computed as follows:

$$10000_{16} + (high - D800_{16}) \cdot 400_{16} + (low - DC00_{16})$$

- Thus the pair (D800,DC00) has code point 65536 and pair (DBFF, DFFF) has code point 1114111

#### characters

- Typically we use a single accumulator like a `StringBuilder` to get the character data in an element content without mixed content. In a document with mixed content, we have to use multiple accumulators in a stack
- When we reach the end-tag, we know we have all the characters
- Do we get the ignorable whitespaces (indentation) ?
  - If the parser is validating: no. They are received through `ignorableWhitespace`
  - If the parser is not validating: yes or no. Check the implementation documentation

### 6.5.4 ContentHandler ignorableWhitespace

#### ignorableWhitespace

```
public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException
```

- The parser reports a chunk of whitespaces with this method going from `ch[start]` to `ch[start+length-1]`
- Contiguous characters may be reported by multiple calls. But all the characters in a given call must come from the same *external* entity
- See section 2.10 of the XML rec. 5th edition

## 6.6 Example 3: handling errors

### SAXExample3.java

- We define an error handler class

#### SAXExample3 on a valid book.xml

```
mac:sax ludo$ java SAXExample3 ../dtd/book.xml
Everything is OK.
```

#### SAXExample3 on a not well-formed book.xml

```
mac:sax ludo$ java SAXExample3 ../dtd/book.xml
*** fatalError called !
The end-tag for element type "title" must end with a '>' delimiter.
Handler in main. Parsing error: The end-tag for element type "title" must end with a '>'
delimiter.
```

### SAXExample3.java

#### SAXExample3 on an invalid book.xml: we throw

```
mac:sax ludo$ java SAXExample3 ../dtd/book.xml
*** error called !
The content of element type "book" must match "(isbn,title,author+,publisher?,edition)".
Handler in main. Parsing error: The content of element type "book" must match "(isbn,title,
author+,publisher?,edition)".
```

#### SAXExample3 on an invalid book.xml: we don't throw

```
mac:sax ludo$ java SAXExample3 ../dtd/book.xml
*** error called !
The content of element type "book" must match "(isbn,title,author+,publisher?,edition)".
Everything is OK.
```

### defining ErrorHandler methods

```
public void method(SAXParseException exception) throws SAXException
```

- We will implement the 3 methods of the `ErrorHandler` interface to intercept errors
- `warning` to report problems that are not errors. Implementation dependent and mostly ignored
- `error` for errors that are not fatal like validity constraints violation or use of an undeclared namespace prefix
- `fatalError` for errors that cause the parsing to stop<sup>1</sup>. Typically well-formedness errors

<sup>1</sup>In fact, parsing may continue but only to report additional errors, not data.