XML

A comprehensive introduction

José Santos

November 2010

1 Basics

1.1 What is XML?

So what is XML?

- XML stands for eXtensible Markup Language
- XML was design to carry data, no to display data
- XML tags are not predefined. You must define your own tags
- XML is designed to be self-descriptive
- XML is a W3C Recommendation

The main advantages of XML:

- XML makes data exchange easy. In XML, the data and the descriptions of how it should be structured (the markup) are stored as text that you can easily read and edit.
- The XML markup can be customized and standardized. Moreover, XML documents are self-describing. Each XML element has a tag and you can quickly figure out what this data represents even years later. This means that XML documents are self-documenting.
- XML has a syntax that allows the author of the XML document to give structure to the data. XML is simple in design but can represent complex data.

Or, putting it more concisely:

- XML helps separating content from presentation.
- XML simplifies data sharing. XML data is stored is plain text format, so it is easier to share.

- XML simplifies data transport.
- XML simplifies platform changes.
- XML makes your data more available.
- XML can be used to create new languages.

XML alows users to define tags to indicate structure. An XML document does not provide any instructions on how it is to be displayed. This information is to be included separately in a stylesheet. You can specify a stylesheet using XSL (XML Stlylesheet Language), which is used to translate XML data to HTML for presentation.

XML comprises a whole host of specifications that complement it. We will focuse on two particularly useful specifications: DTDs and XML Schema. These specifications allow you to describe the structure an XML document should take. Basically, XML gives you the ability to describe "semistructured" data. This means that XML by itself, allows you to describe "schema-less" or "self-describing" data. However, when you specify an XML Schema, your are specifying the the structure of the XML documents that you will consider.

1.2 XML elements

XML documents form a tree structure that starts at "the root" and branches to "the leaves".

The basic component in XML is the *element*. This is a piece of text bounded by matching tags such as ⟨tagname⟩ and ⟨/tagname⟩. Inside these tags an element may contain "raw" text, other elements or a combination of the two.

⟨tagname⟩ is called a start-tag and ⟨/tagname⟩ is called an end-tag. Start- and end- tags are also called markups because they markup or explain the data. One of the rules of XML is that these tags must be balanced. This means that they should be closed in the inverse order in which they are opened, like parentheses. There are no predefined tags.

The element that surrounds all the other elements is referred to as the *root element*. All documents must have a single root element. We use repeated elements with the same tag to represent collections. An element can also be empty and an *empty element* may be abbreviated. This is done by putting a '/' at the end of its tag. It is important to stress that all xml documents must have a root element.

XML tags are case sensitive. The tag ⟨letter⟩ is different from the tag ⟨/Letter⟩.

1.3 XML Attributes

XML allows us to associate atributes with elements. There are differences between attributes and tags. A given attribute may only occur once within a tag, while subelements with the same tag may be repeated. The value associated with an attribute must be a string (thus it must necessarily appear between quotes) while an element may also have subelements as well as values. Therefore, attributes should never be used when a piece of data could be represented as a collection.

There are no rules about when to use attributes or when to use elements. In principle, one should use elements instead of attributes, because:

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

Attributes are difficult to read and maintain. Thus, one should use elements for data and attributes for information that is not relevant to the actual data. One should primarily use attributes for metadata, that is, data about data.

1.4 XML Names

The rules for XML element names are also the rules for XML attriubte names, as well as for the names of serveral less common constructs. Collectively, these are referred to as simply *XML names.* XML names may include the following three punctuation marks: the underscore, the hyphen and the period. XML names may not contain other punctuation characters such as quotation marks, apostrophes, dollar signs, carets, percent symbols and semicolons. The colon is allowed but its use is reserverd for namespaces. XML names may not contain any white space of any kind. XML names may only start with letters, idiograms, or the underscore character. They may not start with a number, hyphen, or period.

1.5 References

The character data inside an element must not contain a raw unescaped opening angle bracket (⟨), since this character is always interpreted as the beginning of a tag. If you need to use this character in your text you can escape it using the *entity reference* **&lt;**, the *numeric character reference* **&#60;**, or the *hexadecimal character reference* **&#x3C;**.

```
<script language="javascript">
  if(location.host.toLowerCase().indexOf("ibiblio") &lt; 0){
     location.href = "http://ibiblio.org/xml";
  }
</script>
```

Character data may not contain a raw ampersand (&) either, because it is always interpreted as beginning an entity reference. However, the ampersand may be escaped using the **&amp;** entity reference like this:

```
<company> W.L. Gore \&amp; Associates </company>
```

XML predefines exactly five entity references. These are:

- **&lt;** represents the less-than sign.
- **&amp;** represents the ampersand.
- **&gt;** represents the greater-than sign.
- **&quot;** represents a straight-double quotation mark.
- **&apos;** represents an apostrophe (aka the straight single quote).

```
<img src='oreilly_koala3.gif' width='122' height='66'
                    alt='Powered by O&apos;Reilly books'>
```

1.6 CDATA Sections

The more sections of literal code a document includes and the longer they are, the more tedious using special encoding becomes. Instead, you can enclose each sample of literal code in a *CDATA* section. A **CDATA** section is set off by ⟨![*CDATA*[and]]⟩. Everything inside a CDATA section is treated as raw character data. Less-than signs don't begin tags. Ampersands dont't start entity references. Everything is simply character data, not markup. The only thing that cannot appear in a CDATA section is the CDATA end delimeter:]]⟩.

1.7 Comments

Comments are written as follows:

```
<!-- This a stupid comment -->
```

1.8 Processing instructions

Comments may appear anywhere in the character data of a document. They may also appear before or after the root element. Comments are not elements, so this does not violate the tree structure or the one-root element rules for XML. However, comments may not apapear inside a tag or inside another comment.

XML provides the *processing instruction* as an alternative means of passing information to particualr applications that may read the document. A processing instruction begins with ⟨? and finishes with ?⟩. Immediately following the ⟨? is an XML name called the *target*, possibly the name of the application for which this processing instruction is intended.

1.9 XML declaration

XML documents should (but not have to) begin with an *XML declaration* like the one presented below:

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

The actual characters in the XML document are stored as numerical codes. The most common set is American Standard Code for Information Interchange (ASCII). ASCII codes extend from 0 to 255 to fit within a single byte. However, many scripts are not handled in ASCII, these include Japanese, Arabic, Hebrew, Bengali and many other languages. For this reason the default character set specified for XML by W3C is Unicode not ASCII. UTF-8 is a compressed version of Unicode that uses 8-bit to represent characters.

The **standalone** attribute specifies if the application is required to read any external DTD. In the case of the example above, since it is set to "yes", an application that reads this XML document knows that it must be validated against a DTD which is not included. The **standalone** attribute is optional in an XML declaration. If it is omitted, then the value "no" is assumed.

2 Document Type Definitions (DTDs)

A *document type defintion* (DTD) are a wy to specify an XML language. DTDs are written in a formal syntax that explains precisely which elements may appear where in the document and what the elemnt's contentes and attributes are.

A valid document includes a *document type declaration* that identifies the DTD that the document satisfies. The DTD lists all the elements, attributes, and entities the document uses and the context in which it uses them.

There are however many things that the DTD does not say. In particular, it does not say:

- What the root element of the document is.
- How many instances of each kind of element appear in the document.
- What the character data inside the elements look like.
- The semantic meaning of an element.

Understand that *validity* is optional, whereas well-formedness is mandatory.

Consider the following example:

```
<!ELEMENT person(name, profession*)>
<!ELEMENT name (first_name, last_name)>
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

The DTD is generally stored in a separate file from the documents it describes. This allows it to be easilly referenced from multiple XML documents. However, it can be included inside the XML document if that's convenient. The **.dtd** extension is fairly standad although not specifically required by the XML specification. The MIME media type is: **application/xml-dtd**. Below we present an XML document that respects the DTD shown above:

```
<person>
  <name>
    <first_name>Maria</first_name>
    <last_name>Fragoso</last_name>
  </name>
  <profession>Assistente de Borde</profession>
  <profession>Professora de FrancÃªs</profession>
</person>
```

Observe that the following document is not valid:

```
<person>
  <profession>Assistente de Borde</profession>
  <name>
    <first_name>Maria</first_name>
    <last_name>Fragoso</last_name>
  </name>
</person>
```

because the profession comes before the name.

In order to say that an element can only contain text we use the DTD special keyword #PCDATA which stands for *parsed character data*.

A *valid* document contains a reference to the DTD to which it must be compared:

```
<!DOCTYPE person SYSTEM "http://www.somewhere.com/person.dtd">
```

This says that the root element of the document is person and that the DTD for this document can be found in the specified URL. The document type declaration is included in the prolog of the XML document after the XML declaration but before the root element.

The DTD can be specified inside the document itself in the following way:

```
<!DOCTYPE person[
  <!ELEMENT person (name, profession*)>
  <!ELEMENT name (first_name, last_name)>
  <!ELENENT first_name (#PCDATA)>
  <!ELEMENT last_name (#PCDATA)>
  <!ELEMENT profession (#PCDATA)>
]>
```

Rules to specify a DTD:

- Element declaration:

  ```
  <!ELEMENT name content_specification>
  ```

- #PCDATA declaration:

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

- Child elements specification:

  ```
  <!ELEMENT fax (phone_number)>
  ```

- Sequences (the corresponding elements must appear in the specified order):

  ```
  <!ELEMENT name (first_name, last_name)>
  ```

- Arbitrary number of children. You can affix one of three suffixes to an element name in a content specification to indicate how many of that element are expected at that position:

  - ?: zero or one element is allowed.
  - *: zero or more elements are allowed.
  - +: one or more elements are allowed.

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

- Choices.

  ```
  <!ELEMENT methodRequest (params | fault)>
  <!ELEMENT digit (zero | one | two | ... )>
  ```

- Mixed Content. You can add any number of child elements to the list of mixed content, but #PCDATA must always come first!!!!!

  ```
  <!ELEMENT paragraph
     (#PCDATA | name | profession | footnote | emphasize | data)*
  >
  ```

- Empty elements. If an element is supposed to be empty. Just write it!!!!

  ```
  <!ELEMENT image EMPTY>
  ```

- Attribute list specification:

  ```
  <!ATTLIST sku
     list_price             CDATA #REQUIRED
     suggested_retail_price CDATA #IMPLIED
     actual_price           CDATA #IMPLIED
  >
  ```

There are 10 attribute types in XML. They are:

- **CDATA:** A CDATA attribute value can contain any string of text acceptable in a well-formed XML attribute value.
- **NMTOKEN:** An XML *name token* is very close to an XML name (it includes more things).

  ```
  <!ATTLIST journal year NMTOKEN #REQUIRED>
  ```

- **NMTOKENS:** Contains one or more XML name tokens separated by whitespace.

  ```
  <!ATTLIST performances dates NMTOKENS #REQUIRED>
  ```

- **Enumeration:** List of possible values for the attribute separated by vertical bars. Each possible value must be an XML name token.

  ```
  <!ATTLIST date month (January | February | Mars | April | May | June
      | July | August | September | October | November | December) #REQUIRED
  >
  ```

- **ID:** An ID type attribute must contain an XML name (not a token but a name) that is unique within the XML document. More precisely, no other ID type attribute in the document can have the same value. Each element may have no more than one ID type attribute.

  ```
  <!ATTLIST employee social_security_number ID #REQUIRED>
  ```

  Note that ID numbers pose us a problem beacause a number is not an XML name and therefore not a legal XML ID. The normal solution is to prefix such ID values with underscore or a common letter.

- **IDREF:** An IDREF type attribute refers to the ID type attribute of some element in the document. Thus, it must be an XML name. IDREF attributes are commonly used to establish relationships between elements when simple containment won't suffice. Consider the following XML document:

  ```
  <project id="p1">
    <goal>Develop a really good thing</goal>
    <team_member person="ss1"/>
    <team_member person="ss2"/>
  </project>

  <employee social_security_number="ss1">
    <name>Manel</name>
  </employee>
  ```

```
<employee social_security_number="ss2">
  <name>Jaquim</name>
</employee>
```

The corresponding can be given by:

```
<!ELEMENT project (goal, team_member*)>
<!ELEMENT team_member EMPTY>
<!ELEMENT goal (#PCDATA)>
<!ELEMENT employee (name)>
<!ATTLIST project id ID #REQUIRED>
<!ATTLIST team_member person IDREF #REQUIRED>
<!ATTLIST project id ID #REQUIRED>
```

- **IDREFS:** An IDREFS type attribute contains a whitespace-separated list of XML names, each of which must be the ID of an eleemnt in the document.

```
<!ATTLIST employee social_security_number ID #REQUIRED>
<!ATTLIST project id ID #REQUIRED
                  team IDREFS #REQUIRED
>
```

- **ENTITY:** An ENTITY type attribute contains the name of an unparsed entity declared elsewhere in the DTD.
- **ENTITIES:** An ENTITIES types attribute contains the name of one or more unparsed entities declared elsewhere in the DTD, separated by whitespace.
- **NOTATION:** A NOTATION type attribute contains the name of a notation declared in the document's DTD.

In addition to providing a data type, each ATTLIST declaration includes a default declaration for that attribute:

- **#IMPLIED:** The attribute is optional. Each instance of the element may or may not provide a vlaue for the attribute. No default value is provided.
- **#REQUIRED:** The attribute is required. Each instance of the element must provide a value for the attribute.
- **#FIXED:** The attribute value is constant and immutable. This attribute has the specified value regardless of whether the attribute is explicity noted on an individual instance of the element.
- **Literal:** The actual value is given as a quoted string.

2.1 Entity Declarations

XML predefines five entities: **&lt;**, **&gt;**, **&amp;**, **&quot;** and **&apos;**. The DTD can define many more entities. Entity references are defined with an *ENTITY declaration* within the DTD. This gives the name of the ENTITY, which must be a XML name, and the replacement text of the entity.

```
<!ENTITY super "supercalifragilisticexpi...">
```

General entities insert replacement text into the body of an XML document. They can also be used inside the DTD in places where they will eventually be included in the body of XML document, for instance in an attribute default value, or in the replacement text of another entity. However, they cannot be used to provide the text of the DTD itself. For instance:

```
<!ENTITY xuxu "xiribitatataUrra!!!">
<!ELEMENT xixi (xuxu)>
```

This is not a legal DTD!!!!

An *external parsed general entity* reference is declared in the DTD using an ENTITY declaration. However, instead of the actual replacement text, the **System** keyword and the URL to the replacement text are provided. For example:

```
<!ENTITY footer SYSTEM "http://www.xixi.pt/susy.xml">
```

When the general entityt reference **&footer;** is seen the parser replaces it with the document available at the provided URL. References to external parsed entities are not allowed in attribute values. Addtionally, observe that the external entity itself must be well-formed. In particular, the external entity may not have a single root element. However, if such a root element were wrapped around the external entity then the resulting document should be well-formed.

An *external unparsed entity* is declared in the DTD using the ENITY declaration in the following way:

```
<!ENTITY turing_get_off_bus SYSTEM
     SYSTEM "http://www.turing.com/bus.jpg"
     NDATA jpeg>
```

2.1.1 Parameter entities

It is not uncommon for multiple elements to share all or part of the same attribute list and content specification. For example:

```
<!ELEMENT apartment (address, footage, rooms, baths, rent)>
<!ELEMENT condo (address, footage, rooms, baths, price)
...
```

An entity reference is the obvious candidate here. However, general entity references are not allowed to provide replacement text for a content specification or attribute list, only for parts of the DTD that will be included in the XML document itself. Instead, XML provides a new construct exclusively for use inside DTDs, the *parameter entity*, which is referred to by a *parameter entity reference*. Parameter entities behave and are declared almost exactly like a general entity. However, they use a % instead of an &, and they can only be used in a DTD, while general entities can only be used in the document content. Example:

```
<!ENTITY % residential_content "address, footage, rooms, baths">
<!ENTITY % rental_content "rent">
<!ENTITY % pruchase_content "price">
```

Parameter entities are dereferenced in the same way as a general entity reference, only with a percent sign instead of an ampersand:

```
<!ELEMENT apartment (%residential_content;, %rental_content;)>
<!ELEMENT condo (%residential_content;, %purchase_content;)>
```

3 Namespaces

Namespaces have two purposes in XML:

- To distinguish between elements and attributes from different vocabularies with different meanings that happen to share the same name.
- To group all the related elements and attributes from a single XML application together so that software can easily recognize them.

The need for namespaces arise from the fact that some documents combine markup from multiple XML applications.

Namespaces distinguish between elements with different meanings but the same name by assigning each element an URI. Generally, all the elements from one XML application are assigned to one URI, and all the elements from a different XML application are assigned to a different URI. These URIs are called namespace names. Elements with the same name but different URIs are different kinds of elements. Elements with the same name and the same URI are the same kind of element. Most of the time a single XML application has a single namespace URI for all its eleemnts, though a few applications use multiple namespaces to subdivide different parts of the application.

Since URIs frequently contain characters that cannot be used in legal XML names, short prefixes such **rdf** and **xsl** stand in for them in element and attribute names. Each prefix is associated with a URI. Names whose prefixes are associated with the same URI are in the same namespace. Names whose prefixes are associated with different URIs are in different namespaces. Prefixed elements and attributes in namespaces have names that contain exactly one colon. They look like this:

```
rdf:description
xlink:type
xsl:template
```

Everything before the colon is called the *prefix.* Everything after the colon is called the *local part.* The complete name, including the colon, is called the *qualified name*, or *raw name.* The prefix identifies the namespace to which the element or the attribute belongs.

Each prefix in a qualified name must be associated with a URI. Prefixes are bound to namespace URIs by attaching an **xmlns:prefix** attribute to the prefixed element or one of its ancestors. For example, the **xmlns:rdf** attribute of the **rdf:RDF** element binds the prefix rdf to the namespace URI http://www.xxx.com/tralala#.

```
<rdf:RDF xmlns:rdf="http://www.xxx.com/tralala#">
 <rdf:Description>
   <title>Impressionist Paintings</title>
   <creator>Elliot Rusty Harold</creator>
   <description>
      A list of famous impressionist paintings organized by
      painter and date.
   </description>
 </rdf:Description>
</rdf>
```

Bindings have scope within the eleemnt where they're declared and within its contents. The **xmlns:rdf** attribute declares the **rdf** prefix for the **rdf:RDF** element as its descendant elements.

It is possible to redefine a prefix within a document so that in one element the prefix refers to one namespace URI, while in another it refers to a different namespace namespace URI. In this case, the closest ancestor element that declares the prefix takes precedence.

Before a prefix can be used, it must be bound to a URI like above. It is the URI that are standardized, not the prefixes. The prefix can change as long as the URI stays the same. Namespace URIs do not necessarily point to any actual document page. In fact, they don't have to use the http scheme. However, if you're defining your own namespace using an http URI, it would not be a bad idea to place some documentation for the XML application at the namespace URI.

You can indicate that an uprefixed eleemnt and all its unprefixed descendants belong to a particular namespace by attaching an xmlns attribute with no prefix to the top element as follows:

```
<svg xmlns="uri-that-you-want-to-use">
 lot's of things
</svg>
```

Note that an XML parser that does not know about namespaces should not have any trouble reading a document that uses namespaces. Colons are legal characters in XML element and attribute names. The parser will simply report that some of the names contain colons. A namespace-aware parser does add a couple of checks to the normal well-formedness checks that a parser performs. Specifically, it checks to see that all prefixes are mapped to URIs. It will reject documents that use unmapped prefixes. It will further reject any element or attribute names that contain more than one colon.

Namespaces do not in any way change DTD syntax nor do they change the definition of validity. For instance, the DTD of a valid document that uses an element named **dc:title** must include an ELEMENT declaration properly specifying the content of the **dc:title** element. The name of the element in the document must exactly match the name of the element in the DTD, including the prefix. The DTD cannot omit the prefix and simply declare a title element. The same is true with respect to prefixed attributes.

We can use parameter entities to deal with this problem. The trick is to define both the namespace prefix and the colon that separtes the prefix from the local name as parameter entities like this:

```
<!ENTITY % dc-prefix "dc">
<!ENTITY % dc-colon ":">

<!ENTITY % dc-title "%dc-prefix;%dc-colon;title">
<!ENTITY % dc-creator "%dc-prefix;%dc-colon;creator">
<!ENTITY % dc-description "%dc-prefix;%dc-colon;description">
<!ENTITY % dc-date "%dc-prefix;%dc-colon;date">

<!ELEMENT %dc-title; (#PCDATA)>
<!ELEMENT %dc-creator; (#PCDATA)>
<!ELEMENT %dc-description; (#PCDATA)>
<!ELEMENT %dc-date; (#PCDATA)>
```

Note that if you want to use the default namespace instead, you only have to change the two first lines:

```
<!ENTITY % dc-prefix "">
<!ENTITY % dc-colon "">
```

4 Internationalization

XML documents contain Unicode text. Unicode is a character set large enough to include all the world's living languages and a few dead ones. It can be written in a variety of encoding, including UCS-2 and the ASCII superset UTF-8.

The encoding declaration specifies which *character set* a document uses. A *character set* maps particular characters to particular numbers. These numbers are called *code points*. A *character enconding* determines how these conde points are represented in bytes. A *character set* like Unicode may then be encoded in multiple encodings, such as UTF-8, UCS-2, or UTF-16. However, most simpler character sets, such as ASCII only have one encoding.

Some environments keep track of which encodings particular documents are written in. For instance, web servers that transmit XML documents precede them with an HTTP header that looks something like this:

```
Content-Type: text/xml; charset=iso-8859-1
```

The **Content-Type** field of the HTTP header provides the MIME media type of the document. This may, as shown here, specify which character set the document is written in. Many web servers omit the **charset** parameter from the MIME media type. In this case, if the MIME media type is **media/xml**, then the document is assumed to be in the US-ASCII encoding. If the MIME media type is **application/xml**, then the parser tends to guess the character set by reading the first few bytes of the document.

Every XML document should have an *encoding declaration* as part of its XML declaration. The *enconding declaration* tells the parser in which character set the document is written. For example:

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes">
```

Even if metadata is not available the encoding declaration can be omitted if the document is written in either UTF-8 or UTF-16 encodings of Unicode. UTF-8 is a strict superset of ASCII, so ASCII files can be legal XML documents without an encoding declaration.

5 XSL Transformations

The Extensible Stylesheet Language (XSL) is divided into two parts: XSL Transformations (XSLT) and XSL Formatting Objects (XSL-FO).

XSLT is an XML application for specifying rules by which one XML document is transformed into another XML document. An XSLT stylesheet contains a list of *template rules*. Each *template rule* has a *pattern* and a *template*. A XSLT processor compares the elements and other nodes in an input XML document to the template-rule patterns in the stylesheet. When one matches, it writes the template from that rule into the output tree. When it's done, it may further serialize the ouput tree into an XML document or some other format like plain text or HTML.

An XSLT stylesheet is an XML document. It can and generally should have an XML declaration. the root element of this document is either **stylesheet** or **transform** (which are synonyms).

Throughout this section we will use the following XML document as an example:

```
<?xml version="1.0"?>
<people>
```

```
  <person born="1912" died="1954">
     <name>
       <first_name>Alan</first_name>
       <last_name>Turing</last_name>
     </name>
     <profession>Computer Scientist</profession>
     <profession>mathmatician</profession>
  </person>
</people>
```

XSLT elements are in a namespace which is generally mapped to the **xsl** prefix. XML documents that will be served directly to web browsers can have an xml-stylesheet processing instruction in their prolog telling the browser where to find the associated stylesheet for the document.

```
<?xml-stylesheet type="application/xml" href="the-corresponding-url"?>
```

- The root element is the **xsl:stylesheet** element:

  ```
  <xsl:stylesheet xmlns:xsl="the-appropriate-url" version="1.0">
    ...
  </xsl:stylesheet>
  ```

- Each template rule is represented by an **xsl:template** element.

  ```
  <xsl:template match="...">
    ....
  </xsl:template>
  ```

- The element **xsl:value-of** calculates the string value of an XPath expression and inserts it into the output. The value of an element is the text content of the element after all the tags have been removed and entity and character references have been resolved. The element whose value is taken is the XPath expression.

  ```
  <xsl:value-of select="an-xpath" />
  ```

Complete example:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://url">
  <xsl:template match="person">
    <p> <xsl:value-of select="name" /> </p>
  </xsl:template>
</xsl:stylesheet>
```

By default, an XSLT processor reads the input XML document from top to bottom, starting at the root of the document and working it sway down using preorder traversal. Template rules are activated in the order in which they match elements encountered during this traversal. This means a template rule for a parent will be activated before template rules matching the parent's children. However, one of the things a template can do is change the order traversal. It can specify that an element should be processed in the middle of processing another element. It can even prevent particular elements from being processed.

The **xsl:apply-templates** element makes the processing order explicit. Its **select** attribute contains an XPath expression telling the XSLT processor which nodes to process at that point in the output tree.

```
<xsl:template match="people">
  <xsl:apply-templates select="person"/>
</xsl:template>

<xsl:template match="person">
  <xsl:apply-templates select="name"/>
</xsl:template>

<xsl:template match="name">
  <xsl:value-of select="last-name"/>
  <xsl:value-of select="first-name"/>
</xsl:template>
```

Note that if you would rather apply templates to all types of children of the **people** element, rather than just **person** children, you can omit the select attribute:

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

5.1 The built-in template rules

There are seven kinds of nodes in an XML document: the root node, element nodes, attribute nodes, text nodes, comment nodes, processing instruction nodes and namespace nodes. XSLT provides a default built-in template rule for each of these senven kinds of nodes that says what to do with that node if the stylesheet author has not provided more specific instructions. These rules use special wildcard patterns to match all nodes of a given type.

5.1.1 The default template rule for text and attribute nodes

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

The **text()** node text is a pattern matching all text nodes. The * is a pattern matching all attribute nodes. This template says that whenever a text or attribute node is matched, the processor should output the value of that node. For a text node, this value is simply the text in the node. For an attribute, this value is simply the attribute value but not the name.

5.1.2 The default template rule for element and root nodes

The most important template rule is the one that guarantees that children are processed. Here is that rule:

```
<xsl:template match="*|/">
  <xsl:apply-templates />
</xsl:template>
```

The asterisk * is an XPath wildcard that matches all element nodes, regardless of what name they have or what namespace they're in. The forward slash / is a XPath wildcard that matches the root node. In isolation this rule means that the XSLT processor eventually finds and applies templates to all nodes except attributes and namespace nodes because every nonattribute, non-namespace node is either the root node, a child of the root node, or a child of an element.

5.1.3 The default template rule for comment and processing instruction nodes

```
<xsl:template match="processing-instruction()|comment()" />
```

It matches all comments and preprocessing instructions. However, it does not output anything into the result tree.

5.1.4 The default template rule for Namespace nodes

This is truly a built-in rule that must be implemented in the XSLT processor's source code; it can't even be written down in an XSLT stylesheet because there is no such thing as an XPath pattern matching a namespace node. That is, there is no **namespace()** node test in XPath.

5.2 Modes

Both **xsl:apply-templates** and **xsl:template** elements can have optinal mode attributes that connect different template rules to different positions. A *mode attribute* on an **xsl:template** element identifies in which mode that template rule should be activated. An **xsl:apply-templates** element with a **mode** attribute only activates template rules with matching mode attributes.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="url-that-identifies-the-xsl-namespace">
```

```
<xsl:template match="people">
  <html>
    <body>
       <h1> A web page for the people </h1>
       <h2> TOC </h2>
       <ul>
         <xsl:apply-templates mode="toc" select="person" />
       </ul>
       <br />
       <h2> Our people</h2>
       <xsl:apply-templates select="person"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="person" mode="toc">
  <xsl:apply-templates select="name" mode="toc"/>
</xsl:template>

<xsl:template match="name" mode="toc">
  <li><xsl:value-of select="last_name" />,
      <xsl:value-of select="first_name" />
  </li>
</xsl:template>

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

</xsl:stylesheet>
```

For every mode in the stylesheet, the XSLT processor adds one default template rule to its set of built-in rules. This applies to all element and root nodes in teh specified mode and applies templates to their children in the same mode.

```
<xsl:template match="*|/" mode="toc">
  <xsl:apply-templates mode="toc"/>
</xsl:template>
```

5.3 Attribute value templates

It's easy to include known attribute values in the ouput document as the literal content of a literal result element.

```
<xsl:template match="person">
  <span class="person"><xsl:apply-templates/></span>
</xsl:template>
```

However, if the value of the attribute is not known when the stylesheet is written, but instead must be read from the input document. The solution is to use an *attribute value template*. An *attribute value template* is an XPath expression enclosed in curly braces that's palced in the attribute value in the stylesheet.

Suppose that the desired output is:

```
<name first="Jose" last="Santos" />

<xsl:template match="name">
  <name first="{first_name}" last={last_name}/>
</xsl:template>
```

The value of the first attribute in the stylesheet is replaced by the value of the *first_name* element from the input document.

6 XPath

XPath is a non-XML language for identifying particular parts of XML documents. You can look at XPath as a language for pinking nodes and sets of nodes out of XML trees. From the perspective of XPath there are seven kinds of nodes:

- The root node
- Element nodes
- Text nodes
- Attribute nodes
- Comment nodes
- Processing-instruction nodes
- Namespace nodes

The root node of an XML tree is not the same as the root element. The root node of the tree contains the entire *document* including the root element, as well as any comments and processing instructions that occur before the root element start-tag or after the root element end-tag. Note however, that the XPath data model does not incldue everything in the document. In particular, the XML declaration, the DOCTYPE declaration and the various parts of the DTD are not addressable via XPath.

In this section we shall systematically refer to the following example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="application/xml" href="people.xsl"?>
<!DOCTYPE people [
   <!ATTLIST homepage
        xlink:type CDATA #FIXED "simple"
        xmlns:xlink CDATA #FIXED "url">
  <!ATTLIST person id ID #IMPLIED>
]>
<people>
  <person born="1985" id="p324">
     <name>
        <first_name>Jose</first_name>
        <last_name>Santos</last_name>
     </name>
     <profession>Informatico</profession>
     <profession>Professor</profession>
     <homepage xlink:href="mypage"/>
  </person>
  <person born="1989" id="p333">
     <name>
        <first_name>Pedro</first_name>
        <last_name>Cardoso</last_name>
     </name>
     <profession>Arquitecto</profession>
     <profession>Estudante</profession>
     <hobby>Reading</hobby>
  </person>
</people>
```

6.1 Location Paths

The most useful XPath expression is a *location path.* A location path identifies a set of nodes in a document. This set may be empty, may contain a single node, or may contain several nodes. A *location path* is built out of successive *location steps.* Each *location step* is evaluated relative to a particular node in the document called the *context node.*

6.1.1 The root location path

The simplest location path is the one that selects the root node of the document. This is simply the forward slash /. The root location path (/) is said to be an *absolute location path* because no matter what the context node is, it always means the same thing: the root node of the document.

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

6.1.2 Child Elemment location steps

The second simplest location path is a single element name. This path selects all child elements of the context node with the specified name. For example, the XPath **profession** referes to all profession child elements of the context node. In XSLT, the context node for an XPath expression used in the **select** attribute of **xsl:apply-templates** and similar elements is the node that is currently matched.

6.1.3 Attribute location steps

Attributes are addressable by XPath. To select a particular attribute of an element, use an @ sign follwed by the name of the attribute you want. For example, the XPath expression **@born** selects the born attribute of the context node.

```
<?xml version="1.0">
<xsl:stylesheet version="1.0"
   xmlns:xsl="url_that_identifies_the namespace">
<xsl:template match="/">
  <html>
     <xsl:apply-templates select="people"/>
  </html>
</xsl:template>
<xsl:template match="people">
  <table>
    <xsl:apply-templates select="person"/>
  </table>
</xsl:template>
<xsl:template match="person">
  <tr>
     <td><xsl:value-of select="name"/> </td>
     <td><xsl:value-of select="@born"/> </td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

6.1.4 Other location steps

XPath also provides a way to refer to namespace nodes, text nodes, process-instruction nodes and comment nodes. Namespace nodes are handled explicitly. The other types have special node tests to match them:

- **comment()**
- **text()**
- **process-instruction()**

Since comments and text nodes don't have names, the **comment()** and **text()** location stpes match any comment or text node child of the context node.

```
<xsl:template match="comment()">
  <i>Comment deleted</i>
</xsl:template>
```

6.1.5 Wildcards and alternatives

Wildcards match different element and node types at the same time. There are three wildcard:

- The asterisk (*) wildcard matches any element node regardless of name. You can put a prfix before the asterisk (for example: **svg:***). In this case, only elements in the namespace corresponding to the prefix are matched.
- The **node()** wildcard matches not only all element tyeps but also the root node, text nodes, process-instruction nodes, namespace nodes, attribute nodes, and comment nodes.
- The @* wildcard matches all attribute nodes. As with elements, you can attach a namespace prefix to the attribute wildcard to match attributes in a specific namespace. For instance: **@xlink:***.

Alternatives are specified as in the following example:

```
<xsl:template match="first_name|last_name|profession|hobby">
  <xsl:value-of select="text()"/>
<xsl:template>
```

6.2 Compound Location paths

Location paths can be combined with a forwardd slash (/) to make a compound location path. For example:

```
/people/person/name/first_name
```

The evaluation of this location path yields all element nodes that are accessible from the root element in the specified way: if there exist a people element P, a person element p, a name n and first_name fn such that P is a child of the root, p is a child of P, n is a child of p and fn is a child of n, then fn is included in the result of the evaluation of the given XPath expression.

A double slach forwared (//) selects from all descendants of the context node, as well as the context node itself. At the beginning of an XPath, it selects from all of the nodes in the document. For example, the XPath expression **//name** selects all **name** elements in the document. The expression **//@id** selects all attributes nodes name id in the document. The expression **person//@id** selects all the attribute nodes named id contained in an element named person.

A double period (..) can be used to indeicate the parent of the current node. For example, the XPath **//@id** selects all the id attributes in the document. Therefore **//@id/..** identifies the node elements that have an attribute named id. The XPath expression:

```
//middle_nmame/../first_name
```

selects the first_name nodes that are siblings of a middle_name.

Finally, the single period (.) indicates the context node. In XSLT this is most commonly used when you need to take the value of the currently matched context.

```
<xsl:template match="comment()">
  <span class="comment"><xsl:value-of select="."/></span>
</xsl:template>
```

6.3 Predicates

Several examples on the use of predicates in the specification of location paths:

- **//profession[. = "physicist"]**: all element nodes of type profession whose value is physicist.
- **//person[profession = "physicist"]**: all element nodes of type person that have an element child of type profession whose value is physicist.
- **//person[@born¡=1920 and @born¿=1910]**: all element nodes of type person that have an attribute **born** whose value is between 1910 and 1920.
- **//name[first_name="richard" or first_name="rick"]**: all element nodes of type name that have an element node child of type first_child whose value is either "richard" or "rick".
- **//name[2]**: selects the second name element in the document.
- **//name[middle_name]**: all the elements of type name that have a middle_name child element.
- **/people/person[@born ¡ 1950]/name[first_name = "Alan"]**: you guess.

6.4 Unabbreviated location paths

They are not very used in practice. Check the book on page 172.

6.5 General Path expressions

XPath expressions are not obliged to return document nodes. XPath expressions do not have necessarily return a node (or a set of nodes). XPath expressions can also return numbers, booleans and strings. XPath expressions that aren't node sets can't be used in the match attribute of an **xsl:template** element. However, they can be used as values for the select attribute of **xsl:value-of**, as well as in the *location path* predicates.

6.5.1 Numerical expressions

XPath provedes five basic operations on numbers: addition (+), subtraction (-), multiplication (*), division (**div**) and the remainder operation (**mod**). The element:

```
<xsl:value-of select="6*7"/>
```

inserts the number 42 into the output tree where the template is instantiated. Another example:

```
<xsl:template match="person">
  <century>
    <xsl:value-of select="(((@born - (@born mode 100)) div 1000) + 1)"/>th
  <century>
</xsl:template>
```

This rule replaces each person by the century in which he or she was born.

6.5.2 Boolean expressions

XPath provedes all the usual relational operators including =, !=, ¡, ¿, ¡= and ¿=. Note that booleans are commonly used in predicates of location paths. For example, the location step:

```
person[profession="physicist"]
```

uses a boolean expression. Boolean expressions are also used in the test attribute of the **xsl:if** and **xsl:when** elements. The following examples illustrate their use:

```
<xsl:template match="profession">
   <xsl:if test=".='computer scientist' or .='physicist'">
      <xsl:value-of select="."/>
   </xsl:if>
</xsl:template>

<xsl:template match="profession">
  <xsl:choose>
     <xsl:when test=".='computer scientist'">
       <i><xsl:value-of select="."/></i>
```

```
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="."/>
    </xsl:otherwise>
  </xsl:choose>
</xsl>
```

You can also use the function **not()**. Like in:

```
not(.='computer scientist')
```

If you use:

```
<xsl:value-of select="true"/>
```

in an XSLT stylesheet, then the XSLT processor looks for all the child elements of the context node named **true**. However, the XPath functions, **true()** and **false()** can substitute for the missing literals quite easily.

6.5.3 XPath functions

In this section, we will just give some examples of functions that can be used in XPath expression. Naturally, we are not exhaustive.

- **position():** returns the position of the current node in the context node list as a number.
- **last():** returns the number of nodes in the context node list, which happens to be the same as the position of the last node in the list.
- **count():** returns the number of nodes in the node-set argument rather than in the context node list. count(//name) returns the number of name elements in the document.
- **id()**. The id() function takes as an argument a string containing one or more IDs separated by whitespace and returns a *node-set* containing all the nodes in the document that have those IDs. When we say ids we mean attributes delcared to have type ID in the DTD not necessarily attributes named ID or id. The id() function allows you to form absolute location paths that don't start from the root. For example, **id('p344')/name** refers to the name element whose parent is the element whose id is p344.

7 XLinks

XLinks are an XML syntax for describing directed graphs, in which vertices are documents at particular URIs and the edges are the links are the edges between the documents. But what you put in the graph is up to you. This is a very (VERY!) brief review of XLinks.

A *simple link* defines a one-way connection between two resources. The sourece or *starting resource* of the connection is the link element itself. The target or *ending resource* of the connection is identified by a *Uniform Resource Identifier (URI)*. The link goes from the starting resource to the ending resource. The starting resource is always an XML element. The ending resource may be an XML document, a particular element in an XML document, a group of elements in an XML document, or something that isn't a part of an XML document, such as an MPEG movie or a PDF file.

A XLink is encoded in an XML document as an element of arbitrary type that has an **xlink:type** attribute with value **simple** and an **xlink:href** attribute whose value is the URI of the link target. The **xlink** prefix must be mapped to the **http://www.w3.org/1999/xlink** namespace URI.

Every XLink element must have a **xlink:type** attribute indicating the kind of link:

- simple
- extended
- locator
- arc
- title
- resource

The **xlink:href** attribute indentifies the resource being linked to. It always contain an URI.

It's important to note that a XLink merely indicates that there is a connection between two documents, but it's up to the application reading the XLink to decide what to do with a that link. Nevertheless, page authors can offer suggestions to browsers about how links should be handled by using the **xlink:show** and **xlink:actuate** attributes.

The **xlink:show** attribute tells a browser or other application what to do when the link is activated. The **xlink:show** attribute has five possible values:

- **new:** Open a new window and show the contents of the link's URI.
- **replace:** Show the ending resource in the current document.
- **embed:** Embed a picture of the ending resource in the current document at the location of the link element.
- **other**
- **none:** Specify no behavior.

The **xlink:actuate** attribute tells the browser when to show then content associated with a linl. It has four possible values, which suggest when an application that encounter an XLink should follow it:

- **onLoad:** The link should be followed as soon as the application sees it.
- **onRequest:** The link should be followed when the user asks to follow it.
- **other**
- **none**

Finally, XLink elements can have two additional attributes **xlink:title** and **xlink:role** rthat specify the meaning of the connection between the resources. The **xlink:title** attribute contains a small amount of plain text describing the remote resource. The **xlink:role** attribute contains a URI that somehow indicates the meaning of the link.

We illustrate all these in teh following example:

```
<book xlink:type="simple"
  xlink:href="http://www.my-library.com/fantastic-book-edition-1.pdf"
  xlink:title="edition 1"
  xlink:role="http://www.my-page.com"
  xlink:actuate="onRequest"
  xlink:show="replace">
A Fantastic Book
</book>
```

Whereas a simple link describes a single unidirectional connection between one XML element and one remote resource, an extended link describes a collection of resources. Each path connects exactly two resources. An extended link can be viewed as a directed, labeled graph in which the paths are arcs, the documents are vertices and the URIs are labels. We will not present here exted links.

8 XPointers

XPointers are a non-XML syntax for identifying locations inside XML documents. An XPointer is attached to the end of the URI as its fragment identifier to indicate a particular part of an XML document rather than the entire document. XPointer syntax builds on the XPath syntax. To the four fundamental XPath data types - Boolean, node-set, number and string - XPointer adds points and ranges, as well as functions needed to work with these types.

A URL that identifies a document looks something like this:

```
http://java.sun.com:80/products/jndi/index.html
```

The scheme **http** tells you what protocol the application should use to retrieve the document. The *authority*, **java.sun.com:80** in this example, tells you from which host the application should retrieve the document. The path, **/products/jndi/index.html** in this example, tells you which file in the directory to ask the server for.

Frequently, URLs contain fragment identifiers that point to a particular named anchor inside the document the URL locates. This is separated from the path by the octothorpe, #. So, if we add an anchor element:

```
<a name="download"></a>
```

The link **http://java.sun.com:80/products/jndi/index.html/#download** will cause the Web browser to look for the corresponding named anchor and to scroll the browser window down to the position in the document where the anchor with that name is found.

XPointer gives us more expressivity in the specification of this kind of links. In particular, it uses allows us to use the full XPath language. Furthermore, XPointer expands XPath by providing operations to select particular points in or ranges of an XML document that do not necessarily coincide with any node or set of nodes "selectable" with an XPath expression.

The most basic form of XPointer is simply an XPath expression enclosed in the parentheses of **xpointer()**. For example:

```
xpointer(/)
xpointer(//first_name)
xpointer(id('sec-intro'))
xpointer(/people/person/name/first_name/text())
xpointer(//middle_initial[position()=1]/../first_name)
xpointer(//profession[.="physicist"])
```

Not all of these XPointers necessarily refer to a single element. Depending on which document the XPointer is evaluated relative to, an XPointer may identify zero, one, or more than one node.

If you are uncertain whether a given XPointer will locate something, you can back it up with an alternative XPointer:

```
  xpointer(//first_name)xpointer(//last_name)
```

If doesn't find anything, it returns an empty node set. No special whitespace is required between the individual xpointer() parts, although whitespace is allowed.

Obviously, what an XPointer points to depends on which document it's applied to. This document is specified by the URL that XPointer is attached to. For example:

```
http://example.org/people.xml#xpointer(//name[poisition()=1])
```

Points to first **name** element in the document.

In HTML, the URLs used in **a** elements can contain an XPointer fragment like the one above.

9 Cascading Style Sheets

To associate a stylesheet with an XML file use the following processing instruction:

```
<?xml-stylesheet type="text/css" href="recipe.css"?>
```

9.1 Selectors

A introduction to CSS selectors in a hurry.

- The asterisk is the *universal selector*.

  ```
  * {font-size: large}
  ```

- *Descendant rule.* The following rule matches the quantity elements that are descendants of ingredients elements:

  ```
  ingredient quantity {font-size: medium}
  ```

- Immediate Child. The following rule matches the quantity elements that are imediate childs of ingredient elements:

  ```
  ingredient > quantity {font-size: inherit}
  ```

- If two elements are separated by a plus then it will match the cases in which the second element is an immediate sibling of the first element.

  ```
  directions + story {border-top-style: solid}
  ```

- Square brackets show that you select elements with a particular attribute:

  ```
  step[optional] {display: none}
  *[optional]{display: none}
  step[optional="yes"] {display: none}
  recipe[source˜="Anderson"]{font-weigth: bold}
  ```

  The operator $=$ selects elements that contain a given word as a part of the value of a specified attribute.

- Target an element with a given id.

  ```
  step#P833 {font-weight: 800}
  ```

Pseudo-class selectors:

- **first-child:** matches the first child of the named element.

- **link:** Matches the named element if and only if that element is the source of an yet unvisited link.
- **visited:** This pseudo-class applies to all visited links of the specified type.
- **active:** This pseudo-class applies to all elements the user is currently activating (for example, by clicking on the mouse).
- **hover:** This pseudo-class applies to elements on which the cursor is currently positioned but has not yet activated.
- **focus:** This pseudo-class applies to the element that currently has the focus.

```
step:first-child {font-style: italic}
*:link {color: blue; text-decoration: underline}
*:visited {color: purple; text-decoration: underline}
*:activate {color: red}
*:hover {color: green; text-decoration: underline}
*:focus {border: 1px solid red }
```

Pseudo-element selectors match things that aren't actually elements. Like pseudo-class selectors, they're attached to an element selector by a colon. There are four of these:

- **first-letter:** The first-letter pseudo element selects the first letter of an element.
- **first-line:** The first-line pseudo element selects the first line of a block element.
- **before:** The before pseudo-element selects the point immediately before the specified element.
- **after:** The after pseudo-element selects the point immediately after the specified element.

Some examples:

```
story:first-letter {
  font-size: 200%;
  font-weight: bold;
  float: left;
  padding-right: 3pt
}

story:first-line {font-variant: small-caps}

ingredients:before {content: "Ingredients!"}
```

```
step:before {
   content: counter(step) ". ";
   counter-increment: step;
}
```

9.2 The display property

The **display** property is one of the most important CSS properties. This property determines how the element is postioned on the page. There are 18 legal values for the value of this property. We will cover some of them.

- *Inline elements.* Setting the display to inline, the default value, places the element in the next available posittion from left to right.
- *Block elements.* In contrast to inline elements, an element set to **display:block** is separated from its siblings, generally by a line break.
- *List elements.* An element whose display property is set to **list-item** is also formatted as a block-level element. However, a bullet is inserted at the beginning of the block. The **list-style-type**, **list-style-image** and **list-style-position** properities control which character or image is used for a bullet and exactly how the list is indented.

  ```
  step {
    display: list-item;
    list-style-type: decimal;
    list-style-position: inside
  }
  ```

- *Hidden elements.* An element whose display property is set to none is not included in the rendered element the reader sees.

  ```
  story {display: none}
  ```

- *Table elements.* Lots of options (Go to the book!!!!).

10 XML as a data format

10.1 XML and web services

10.1.1 Web:Rest

One of the earliest approaches, and still one of the best, is transmitting XML over HTTP. The server assembles an XML document and sends it to a client just like it sends an HTML file or a GIF image. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<weatherNow xmlns="http://example.com/weatherNow/">
  <temperature>57</temperature>
  <pressure>29.97</pressure>
  <pressureChange>rising</pressureChange>
</weathernow>
```

This simple web-based approach has been gathering supporters under the banner of *Representational State Transfer*. In the REST model, XML exchanges are treated in a very web-like way, using HTTP methods (GET, PUT, POST, DELETE) as verbs, XML documents as messages and URIs to identify the services.

10.1.2 XML-RPC

XML-RPC is a very simple protocol that encodes the method name and arguments as an XML document and transmits it using HTTP POST. The remote server responds with another XML document enconding the method's return value or an error message. The XML-RPC vocabulary defines elements representing six primitive data type.

10.1.3 SOAP

SOAP offers much more flexibility than XML-RPC, but it is much more complex as well. SOAP (formerly, the simple object access protocol) uses XML to encapsulate information being sent between programs. Like XML-RPC, SOAP started out using HTTP POST requests, and this is still the most common way to use SOAP, although other transport protocols are allowed.

SOAP provides three features that differentiate it from plain XML messaging. The first is a structure for messages containing a **SOAP-ENV:Envelope**, an optional **SOAP-ENV:Header** for metadata and a **SOAP-ENV:Body**. The second, now largely deprecated, is a soap encoding. The last feature is an explicit vocabulary for error messages, which are called faults.

Most of what's gained in SOAP beyond ordinary XML is a wrapper structure that lets developers add their own details to the messages. The **SOAP-ENV:Header** element, which can appear as the first child element of **SOAP-ENV:Envelope**, may be used to add extra information to a request, appearing before the body.

When used in an HTTP environment, the request would typically be sent as a POST request form the client, generating the response from the server. SOAP can be used onver a variety of other protocols, provided that all the senders and receivers understand both the protocol being used and as much of the SOAP messages as they need to process the request.

The *Web Services Description Language (WSDL)* can somewhat automate the process of processing SOAP requests and responses. A WSDL document is itself an XML document that describes a SOAP service.

10.2 Developping Record-like XML Formats

The basic steps involved in creating a new XML application are as follows:

- Determine the requirements of the application.
- Look for existing applications that might meet those requirements.
- Choose a validation model.
- Decide on a namespace structure.
- Plan for expansion.
- Consider the impact of the design on application developers.
- Determine how old and new versions of the application will coexist.

Namespace support is of the utmost importance. If instance documents have to be validated, one has to take into consideration that DTDs are not namespace aware, so strategic use of parameter entities can make modification of prefixes much simpler down the road.

11 XML Schema

Although document type definitions can enforce basic structural rules on documents, many applications need a more powerful and expressive validation method. The W3C developed the XML Schema to address this problem. Multiples schemas can be combined to validate documents that use multiple XML vocabularies.

An XML Schema is an XML document containing a formal description of what comprises a valid XML document. An XML document described by a Schema is called an *instance document*. If a document satisfies all constraints specified by a the schema, it is considered to be *schema valid*.

DTDs provide the capability to do the basic validation of the following items in XML documents:

- Element nesting
- Element occurence constraints
- Permitted attributes
- Attribute types and default values

However, DTDs do not provide fine control over the format and data types of element and attribute values. Other than the various special attribute types, once an element is declared to contain character data no limits may be placed on the lenght, type, or format of that content. Schemas can enforce much more specific rules about the contents of elements and attributes than DTDs can.

In addtion of a wide range of simple types, the schema language provides a framework for declaring new data types, deriving new types from old types, and reusing types from other schemas.

Besides simple data types, schemas can place more explicit restrictions on the number and sequence of child eleemnts that can appear in a given location.

Unlike DTDs schemas validate against the combination of the namespace URI and local name, rather than the prefixed name. XML Schema uses namespaces internally for several purposes.

11.1 Schema Basics

We start with a very simple example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="fullname" type="xs:string"/>
</xs:schema>
```

Consider the following *instance document*:

```
<?xml version="1.0"?>
<fullname xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="address-schema.xsd">Scott Means</fullname>
```

The attribute **xsi:noNamespaceSchemaLocation** contains a URL for the schema used to validate the element and states that the elements to validate are not in any namespace. In contrast, we could have used the **xsi:schemaLocation** attribute if the elements to validate were in particular namespace. In this case, the **xsi:schemaLocation** would contain the namespace and the URL of the Schema.

Every schema document consists of a single root **xs:schema** element. This element contains declarations for all elements and attributes that may appear in a valid instance document. Instance elements declared using top-level **xs:element** elements in the schema are considered *global elements* and should be used carefully. If more than one element is declared globally, a schema valid document may not contain the root element you expect. Naming conflicts are another potential with multiple global declarations. When writing schema declarations, it is an error to delcare two things of the same type at the same scope.

The Schema language as special markup for annotations. The major drawback to using XML comments is that parsers are not obliged to keep comments intact when parsing XML documents. Addtionally, giving comments a struture open the possiblity of automatic documentation generation. The schema elements for specifying comments are:

- The element **xs:annotation** corresponds to the whole comment.
- The element **xs:documentation** must be a child of **xs:annotation** and its contents should specify comments intended for human readers.

- The element **xs:appinfo** must be a child of **xs:annotation** and its contents should specify comments intended for machines.

Consider the following example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:annotation>
  <xs:documentation xml:lang="en-US">
    Simple schema example
  </xs:documentation>
</xs:annotation>

<xs:element name="fullname" type="xs:string"/>

</xs:schema>
```

- The schema element **xs:element** corresponds to the declaration of an element.
- Schemas support two different types of content: *simple* and *complex*. Simple content consists of pure text that does not contain nested elements. XML provides several built-in schema types:
  - **anyURI** A Uniform Resource Identifier
  - **base64Binary** Base64-encoded binary data
  - **boolena** May contain true or false, 0 or 1
  - **byte** A signed byte quantity
  - **dateTime** An absolute date and time
  - **duration** A length of time
  - ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS
  - **integer**
  - **language**
  - **Name**
  - **string**
- Attributes are declared using the **xs:attribute** element. Attributes may be declared globally by top-level **xs:attribute** elements or locally as part of a complex type definition that is associated with a particular element.
- Complex types are declared using the **xs:complexType** element. Elements that have attributes are complex types whether or not they only have simple content.

- Simple content is declared using the **xs:simpleContent** element. The **xs:simpleContent** tells the Schema processor that the content of the element is a simple type. To specify what kind of simple content is allowed you have to use the **base** attribute of the **xs:extension** element that must be declared inside the **xs:simpleContent** element and that specifies that you want to extend that particular kind of content. Attribute declaration go inside the **xs:extension** element.

- You can group attribute declarations and give them a "name" using the **xs:attributeGroup**. The **xs:attribute** elements go inside the **xs:attributeGroup** element. You can give the group a name using the **name** attribute. You can use the same attribute delcaration several time using the **ref** attribute of the **xs:attribute** element. In doing so you are saying that the attributes are those declared in the **xs:attributeGroup** element whose **name** attribute coincides with the **ref** attribute you are passing.

- Associating a schema with a particular namespace is extremely simple: add a **targetNamespace** attribute to the root **xs:schema** element. Naturally, in the instance XML documents, the elements that you want to validate must be declared in the same namesapce. You do that using the **schemaLocation** attribute. Notice that the **schemaLocation** attribute value must contain two tokens:

  - The first is the target namespace URI that matches the target namespace of the schema document.
  - The second is the physical location of the actual schema document.

  Note that references within a Schema definition must be prefixed with the appropriate namespaces. So when you reference a particular attribute group you must prefix the name of the group with the name of the namespace.

- The **elementFormDefault** and **attributeFormDefault** attributes of the **xs:schema** element control whether locally declared elements and attributes must be *namespace-qualified* within instance documents. We recall that attributes unlike elements don't inherit the default namespace declared as the value of the **xmlns** attribute. You have to define a prefix (with **xmlns:prefix="...."**) and use it explicitly to namespace-qualify your atttributes.

Now we provide an example which uses all the concepts previously introduced.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www-sop.inria.fr/members/Jose.Santos/wordcards"
  xmlns:wdb="http://www-sop.inria.fr/members/Jose.Santos/wordcards"
  attributeFormDefault="qualified">
```

```
<xs:element name="word">
  <xs:complexType>
     <xs:simpleContent>
       <xs:extension base="xs:string">
          <xs:attributeGroup ref="wd:language"/>
       </xs:extension>
     <xs:simpleContent>
  <xs:complexType>
</xs:element>
<xs:attriubteGroup name="language">
  <xs:attribute name="lang"  type="wdb:language"/>
</xs:attributeGroup>
</xs:schema>
```

The following XML file corresponds to a valid instance of the Schema presented above:

```
<?xml version="1.0"?>
<wdb:word xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www-sop.inria.fr/members/Jose.Santos/wordcards
                wordcards.xsd"
   xmlns:wdb="http://www-sop.inria.fr/members/Jose.Santos/wordcards"
   wdb:lang="en">Bottle</wdb:word>
```

11.2 Defining new types

The schema language allows us to define new types using the following two schema elements:

- **xs:complexType**
- **xs:simpleType**

If a new type is declared globally it needs to be given a name, so that it can be referred from element and attribute declarations within the schema. If a type is defined *inline*, it does need to be named. But since it has no name, it cannot be referenced by other element or attribute declarations.

The schema element for building complex types is the **xs:complexType** element. Complex types may be named using the corresponding **name** attribute. If a complex type is used directly inside an element declaration **xs:element** without receiving a name, it is denoted an *anonymous type*.

We have seen how to specify simple content using the **simpleContent** element. We can specify content using the **sequence** element. The sequence element tells the schema processor that the contained list of elements must appear in the target document in the exact order they are given. Additionally, it is possible to set the minimum and maximum number of times an element may occur at a particular point in a document using **minOccurs** and **maxOccurs** of the **xs:element** element. The default value for both **minOccurs** and **maxOccurs** is 1, if they are not explicitly provided. Therefore, setting **minOccurs** to 0 means that an element can appear 0 or 1 times.

We now provide an example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.tiririca.com/address"
   xmlns:addr="http://www.tiririca.com/address"
   elementFormDefault="qualified">
<xs:element name="address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="fullname">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="first" type="addr:nameComponent"/>
            <xs:element name="middle" type="addr:nameComponent"
                minOccurs="0"/>
            <xs:element name="last" type="addr:nameComponent"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="nameComponent"/>
  <xs:simpleContent>
     <xs:extension base="xs:string">
        <xs:attributeGroup ref="addr:nationality"/>
     </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:attributeGroup name="nationality">
  <xs:attribute name="language" type="xs:string"/>
</xs:attributeGroup>
</xs:schema>
```

The following document is an instance of the schema specified above:

```
<?xml version="1.0"?>
<addr:address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.tiririca.com/address
       address.xsd"
    xsmls:addr="http://www.tiririca.com/address"
    addr:language="en">
  <addr:fullName>
    <addr:first>Jose</addr:first>
    <addr:last>Santos</addr:last>
  </addr:fullName>
</addr:address>
```

Suppose that we add to the specification of the address elemnt the following element:

```
<xs:element name="contacts" type="addr:contactsType" minOccurs="0"/>
```

Now, obviously, the element declaration:

```
<xs:complexType name="contactsType">
  <xs:sequence>
    <xs:element name="phone" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
         <xs:attribute name="number" type="xs:string"/>
      <xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Notice that the **xs:element** declaration for the previous **phone** element contains a complex type definition that only includes a single attribute declaration. Why???? Because this tells the schema processor that the **phone** element may only contain complex content (elements), and since no additional nested element declarations are provided, it must remain empty.

The phone element declaration could be instead rewritten as follows:

```
<xs:element name="phone" minOccurs="0">
  <xs:complexType>
     <xs:complexContent>
       <xs:restriction base="xs:anyType">
          <xs:attribute name="number"  type="xs:string"/>
       </xs:restriction>
     </xs:complexContent>
  </xs:complexType>
</xs:element>
```

The most common reason to use the **xs:complexContent** element is to derive a complex type from an existing type. This example derives a new type by restriction from the built-in **xs:anytype**. **xs:anytype** is the root of all of the built-in schema types and represents an unrestricted sequence of characters and markup. Since the **xs:complexType** indicates that the element can only contain element content, the effect of this restriction is to prevent the element from containing either character data or markup.

11.3 Simple Content

Earlier, the **xs:simpleContent** element was used to declare an element that could only contain simple content:

```
<xs:element name="fullName">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="language" type="xs:language"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

The **xs:simpleType** element can define new simple data types, which can be referenced by element and attribute declarations within the schema.

How to define new simple types??

```
<xs:simpleType name="location">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
```

This new simple type does not restrict in any way the content allowed by the **xs:string** simple type. However, there is indeed a way to implement such a restriction: *facets*. A *facet* is an aspect of a possible value for a simple data type. The following list cover the different facet types that are supported by a schema processor:

- **length**
- **pattern**
- **enumeration**
- **whiteSpace**
- **maxInclusive** and **maxExclusive**
- **minInclusive** and **minExclusive**
- **totalDigits**
- **fractionDigits**

The **whiteSpace** controls how the schema processor will deal with any whitespace within the target data. Whitespace normalization takes place before any of the other facets are processed . There are three possible values for the **whitespace** facet:

- **preserve** Keep all whitespace exactly as it is in the document.

- **replace** Replace occurrences of tab, line feed and carriage return with space characters.
- **collapse** Perform the replace step first, then collapse multiple-space characters into a single space.

The lenght restriction facets are fairly easy to understand. The **length** facet forces a value to be exactly the length given. The **minLength** and **maxLength** facets can set a definite range for the lengths of values of the type given.

```
<xs:complexType name="nameComponent">
 <xs:simpleContent>
   <xs:extension base="addr:nameString"/>
 </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="nameString">
 <xs:restriction base="xs:string">
   <xs:maxLength value="50"/>
 </xs:restriction>
</xs:simpleType>
```

One of the more useful types of restriction is the simple enumeration.

```
<xs:simpleType name="locationType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="work"/>
    <xs:enumeration value="home"/>
    <xs:enumeration value="mobile"/>
  </xs:restriction>
</xs:simpleType>
```

Minimum and maximum values. Four facets control minimum and maximum values:

- **minInclusive**
- **minExclusive**
- **maxInclusive**
- **maxExclusive**

The primary difference between the inclusive and exclusive flavors of the min and max facets is whether the value given is considered part of the set of allowable values.

```
<xs:maxInclusive value="0"/>
<xs:minInclusive value="1"/>
```

There are two facets that contro the lenght and precision of decimal numeric values: **totalDigits** and **fractionDigits**. The **totalDigits** facet determines the total number of digits that are allowed in a complete number. **fractionDigits** determines the number of those digits that must appear to the right of the decimal point in the number.

The **xs:pattern** facet can place very sophisticated restrictions on the format of string values.

```
<xs:simpleType name="ssn">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d\d\d-\d\d-\d\d\d\d"/>
  </xs:restriction>
</xs:simpleType>
```

Schema allows you to specify simple list types that could be declared as possible attribute values: IDREFS, ENTITIES and NMTOKENS. These list types are themselves simple types and may be used in the same places other simple types are used.

```
<xs:simpleType name="nameListType">
  <xs:list itemType="addr:nameString"/>
</xs:simpleType>
```

In some cases it is useful to allow potential values for elements and attributes to have any of several types. The **xs:union** element allows a type to be declared that can draw from multiple type spaces.

```
<xs:attribute name="location">
  <xs:simpleType>
    <xs:union memberTypes="addr:locationType xs:NMTOKEN" />
  </xs:simpleType>
</xs:attribute>
```

11.4 Mixed Content

XML 1.0 provided the ability to declare an element that could contain parsed character data (#PCDATA) and unlimited occurrences of elements drawn from a provided list. The **mixed** attribute of the **complexType** element controls whether character data may appear within the body of the element with which it is associated. Consider the following example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="letter">
   <xs:complexType mixed="true" />
 </xs:element>
</xs:schema>
```

Trying to validate the following document with the above schema produces an error.

```
<letter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="formletter.xsd">
  Hello!
</letter>
```

This is because there's no complex content for the letter element. Setting **mixed** to true is not the same as declaring an element that may contain a string.

We can allow any type of content by using the **xs:any** element.

```
<xs:element name="notes" minOccurs="0">
  <xs:complexType>
     <xs:sequence>
       <xs:any namespace="url" minOccurs="0" maxOccurs="Unbounded"
              processContents="skip"/>
     </xs:sequence>
  </xs:complexType>
</xs:element>
```

Setting the attribute **processContents** to **skip** means that no validation will be performed.

There is also support in schemas to declare that any attribute may appear within a given element. The **xs:anyAttribute** element may include the **namespace** and **processContents** attributes:

```
<xs:anyAttribute namespace="mynamespaceurl" processContents="skip" />
```

11.5 Element positioning

There are three elements to control element positioning in XML Schema:

- **xs:sequence**
- **xs:choice** Only one of the choice nested elements can appear.
- **xs:all** The xs:all element must appear at the top of the content model and can only contain elements that are optional or appear only once. The xs:all construct tells the schema processor that each of the contained elements must appear once in the target document, but can appear in any order.

We provide examples.

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="greeting"/>
      <xs:element name="body"/>
      <xs:element name="closing"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="greeting">
  <xs:complexType mixed="true">
     <xs:choice>
       <xs:element name="hello"/>
       <xs:element name="hi"/>
       <xs:element name="dear"/>
     </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="body">
 <xs:complexType mixed="true">
   <xs:all>
     <xs:element name="item"/>
     <xs:element name="price"/>
     <xs:element name="arrivalDate"/>
   </xs:all>
 </xs:complexType>
</xs:element>
```

Just as the **xs:attributeGroupp** element commonly used attributes to be grouped together and referenced as a unit, the **xs:group** element allows sequences, choices, and model groups of individual element declarations to be grouped together and given an unique name. These groups can then be included in another element-content model using an **xs:group** element with the **ref** attribute set to the same value as the **name** attribute of the source group.

11.6 Using multiple documents

There are three mechanisms that include declarations from external schemas for use within a given schema: **xs:include**, **xs:redefine** and **xs:import**.

```
<xs:include schemaLocation="blabal.xsd"/>
```

Content that has been included using the **xs:include** element is treated as though it were actually a part of the including schema document. But unlike external entites, the included document must be a valid schema in its own right. That means that it must be a well-formed XML document and have an **xs:schema** element as its root element. Also, the target namespace of the include schema msut match that of the including document.

The **xs:redefine** element work much like the **xs:include** element but in this case types from the included schema may be redefined without generating an error from the schema processor.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="myurl"
  xmlns:addr = "myurl"
  attributeFormDefault="qualified" elementFormDefault="qualified">
  ...
<xs:redefine schemaLocation="physical-address.xsd">
  <xs:complexType name="physicalAddressType">
    <xs:complexContent>
      <xs:extension base="addr:physicalAddressType">
        <xs:attribute name="latitude" type="xs:decimal"/>
        <xs:attribute name="longitude" type="xs:decimal"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>
</xs:schema>
```

Using **xs:import**, it is possible to make the global types and elements that are declared by a schema belonging to another namespace accessible from within an arbitrary schema.

```
<xs:import namespace="myadfasdf" schemaLocation="adfa.xsd"/>
```

11.7 Derived complex types

How to derive complex types using existent ones:

- **xs:extension**
- **xs:restriction**

When deriving a new type from an exsiting type, the resulting type is equivalent to appending the contents of the new declaration to the contents of the *base* declaration.

```
<xs:complexType name="mailingAddressType">
  <xs:complexContent>
    <xs:extension base="addr:physicalAddressType">
      <xs:sequence>
        <xs:element name="zipcode" type="xs:string"/>
      </xs:sequene>
    <xs:extension>
  <xs:complexContent>
<xs:complexType>
```

The biggest benefit of this approach is that as new declarations are added to the underlying type, the derived type will automatically inherit them.

12 Programming Models

Programs can look at XML as plain text, as a stream of events, as a tree, or as a serialization of some other structure.

Event-based parsers are the most used parsers for XML. An event-based parser works as follows. As it reads a document, it moves from the beginning of the document to its end. It may pause to retrieve external resources - for a DTD or an external entity, for instance - but it builds an understanding of the document as it moves along. Each time the event-based parser finds an element it triggers an event.

The upside to an event-based API is speed and efficiency. Because event-based APIs stream the document to the client application, your program can begin working with the data from the beginning of the document before the end of the document is seen. Even more important than speed is size. XML documents can be quite large, an event-based API does not need to store all this data in memory at one time.

Pull processing models are an alternative to event-based models. The main difference between them is that the pull processing models rely on the client application to request content from the parser at its own pace.

Tree-based APIs typically present a model of an entire document to an application once parsing has successfully concluded. Working with a tree model has substantial advantages. The entire document is always available, and moving well-balanced portions of a document from one place to another or modifying them is fairly easy. However, tree models also have a few drawbacks. Namely, they can take up large amounts of memory.

12.1 Common processing issues

- What to do with DTDs? DTDs augment a document's infoset with several important properties, including: entity definitions, default attribute values etc. When to do when a link to a DTD is encountered? Should the application try to retrieve the DTD? This poses questions of security, performance etc.
- What to do with whitespaces?
- What to do with entity references. There are three kinds of entity reference:
  - Numeric character references.
  - The five predefined entity references.
  - General entity references defined by the DTD. Again the problem mentioned above.