Web Technology

Chapter: 3

The Client Tier

XML, DTD, XSD, XSLT

What is XML?

- XML: e**X**tensible **M**arkup **L**anguage.
- XML is a standard for creating markup languages which describe the structure of data.
- It is a widely used system for defining data formats.
- XML is cross-platform, independent tool for exchanging data.
- XML is a **framework** for defining markup languages:
  1. There is **no fixed collection of markup tags** - we may define our own tags, tailored for our kind of information
  2. Each XML language is targeted at its own application domain, but the languages will share many features
  3. There is a common set of **generic tools** for processing documents
  4. is a markup language which relies on the concept of rule-specifying tags and the use of a tag-processing application that knows how to deal with the tags.

XML Basics:

- Developed by W3C (World Wide Web Consortium).
- It is the pared-down version of SGML (Standardized Generalized Markup Language).
- Designed especially for web documents
- Allows you to create your own language for displaying documents.
- "***XML is not the replacement of HTML.***"
- XML requires a processing application. There is no XML browser in market yet.
- XML isn't about display – it's about Structure.
- XML file should be saved with the extension "**.xml**".
- You can use any text editor to write XML code. And the output can be viewed in any browser.
- Every XML page consists of processing instruction at the very first line.
- There should be a root element in every XML document, which should be unique.

Rules for writing XML:

- The first line must be **<?XML version = "1.0"?>**.
- Tags are enclosed in angle brackets.

- Tags are case sensitive and must have a matching closing tag.
- Tags may contain attributes in the form **name = “value”**.
- Tags may contain text, other tags, or both. Tags content lies between the starting and ending tag.

Advantages of XML:

- The first benefit of XML is that because you are writing your own markup language, you are not restricted to a limited set of tags defined by proprietary vendors.
- With XML, you can create your own set of tags at your own pace.
- XML allows every person/organization to build their own tag library which suits their needs perfectly.
- XML allows you to define all sorts of tags with all sorts of rules, such as tags representing business rules or tags representing data description or data relationships.
- With XML, GUI is extracted. Thus, changes to display do not require futzing with the data. Instead separate style-sheet will specify a table display or a list display.
- Searching the data is easy and efficient. Search-engine can simply parse the description-bearing tags rather than muddling in the data. Tags provide the search engines with the intelligence they lack.
- Complex relationships like tree and inheritance can be communicated.
- The code is much more legible to the person coming into the environment with no prior knowledge.

Disadvantages of XML:

- XML requires a processing application.
- There are no XML browsers on the market yet. Thus, XML documents must either be converted into HTML before distribution or converting it to HTML on-the-fly by middleware.
- Barring translation, developers must code their own processing applications.
- XML isn't about display -- it's about structure.
- This has implications that make the browser question secondary. So the whole issue of what is to be displayed and by what means is intentionally left to other applications.

XML Declaration:

- To begin an XML document, it is a good idea to include the XML declaration as the very first line of the document.
- The XML declaration is a processing instruction that notifies the processing agent that the following document has been marked up as an XML document.
- The XML declaration looks as following:

  <?xml version = "1.0"?>
- In its full regalia, the XML declaration might look like the following:

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

Elements:

- Elements are the basic unit of XML content.
- Syntactically, an element consists of a start tag, and an end tag, and everything in between.
- *Example:*

  <NAME>Frank Lee</NAME>
- All XML documents must have at least one root element to be well formed. The root element, also often called the document tag, must follow the prolog (XML declaration plus DTD) and must be a nonempty tag that encompasses the entire document.
- XML defines the text between the start and end tags to be "character data" and the text within the tags to be "markup".

Rules for Elements:

- Every starting tag must have a matching end tag.
- Tags can’t overlap.
- XML documents can have only one root element.
- Element names must XML naming conventions.
- XML is case sensitive.
- XML will keep white space in your text.

Attributes:

- Attributes are properties that are associated with an element.
- An attribute specification is the "*name-value*" pair that is associated with the element.
- Adding attribute provides an alternative way to include information in an element.
- Like XML tags, attributes are also not restricted to store the type of information.
- They are attached with a starting tag.
- Attributes must have values.
- We can use double quote (" ") or single quote (' ') to delineate attribute values.
- *Example:*

  <ELEMENT_NAME attribute_name = "value"> or

  < ELEMENT_NAME attribute_name = 'value'>

Why use Attribute?

- Attributes provide meta-data that may not be relevant to most applications dealing with our XML.
- Attributes are easier to use, they don't require nesting and you don't have to worry about crossed tags.
- Attributes takes much less space than elements.

Rules for Creating Attributes:

- The name must begin with a letter or an underscore ('_'), followed by '0' or some letters, digits, periods ('.'), hyphens('-').
- The XML specification states that attribute names beginning with the prefix "xml" (in any combination of uppercase or lowercase letters) are "reserved for standardization". Although IE-5 doesn't enforce this restriction, its better not to use this prefix to avoid future problems.
- A particular attribute name can appear only once in the same start-tag or empty tag. Example:

  <ANIMATION FileName="a.ani" **FileName**="b.ani"**>** Some Text **</ANIMATION>**
- The value of attribute must be delimited using either single quotes (' ') or double quotes (" ").
- The value of attribute cannot contain the same quote character used to delimit it.
- The value of attribute cannot include the '<'5character and '&' character except to begin a

character or entity reference.

Comments:

- Comments provide a way to insert text that isn't really part of the document, but rather is intended for people who are reading the XML source itself.
- Comments Start with the String <!- - and end with the string - ->
- *Example:*

  <NAME NickName = "Maddy"**>**

  <FIRST-NAME> John **<FIRST-NAME>**

  <MIDDLE-NAME> </MIDDLE-NAME>

  <!- - John lost his Middle Name in Fire - - >

  <LAST-NAME> Doe **</LAST-NAME>**

  </NAME>

- ***Points to Ponder:***
  - You cannot have comment inside a tag.

    <MIDDLE-NAME> </MIDDLE-NAME<!- - John lost his Middle Name in Fire - - > **>**
  - You can't use '- -' character inside a comment.

Escaping Characters:

- There are some reserved characters that you can't include in your PCDATA because they are used in XML syntax.
- For Example:

  '**<**' and '**&**' characters are example of those characters.

  <COMPARISON> 6 is < 7 & 7 > 6 **</COMPARISION>**
- To escape such characters, you simply can use &lt; and &amp; characters.
- It automatically un-escapes the characters for you when it displays the document.

Character Reference:

- The strings such as &#NNN; (where 'NNN' is the Unicode numbers), used to escape the reserved words of XML is known as C*haracter Reference*.
- It can also be &#Xnnn; with 'X' preceding the number and 'nnn' is a hexadecimal number.

Entity Reference:

- The strings such as &lt; used to escape the reserved words of XML is known as *Entity Reference*.
- The Entity Reference can be like &gt; &amp; &lt; etc.
- You can either use character reference or entity reference to escape the characters.

CDATA:

- If you have a lot of '<' and '>' characters that you need to be displayed in XML document, then it is wise to use CDATA sections.
- CDATA is another inherited term from SGML. It stands for Character Data.
- In the special case of CDATA blocks, all tags and entity references are ignored by an XML processor that treats them just like any old character data.
- Using CDATA sections; we can tell the XML parser not to parse the text, until it gets the end of the section.
- CDATA blocks have been provided as a convenience measure when you want to include large blocks of special characters a character data, but you do not want to have to use entity references all the time.
- To avoid the inconvenience of translating all special characters, you can use a CDATA block to specify that all character data should be considered character data whether or not it "looks" like a tag or entity reference.
- *Where to place the CDATA section:*

    - CDATA should not be placed above the root element.
    - CDATA should not be placed within the content of document element.

- *Example of CDATA:*

<EXAMPLE>

```
<![CDATA[
  <COMPARISION>
            6 is < 7 & 7 > 6.
  </ COMPARISION >
]]>
</EXAMPLE>
```

Processing Instructions:

- A processing instruction is a bit of information meant for the application using the XML document.
- That is, they are not really of interest to the XML parser.
- Instead, the instructions are passed intact straight to the application using the parser.
- The application can then pass this on to another application or interpret it itself.
- All processing instructions follow the generic format of:

  <?NAME_OF_APPLICATION Instructions?>
- Example of processing instructions:

```
<?XML version = "1.0"?>
  <doc>
       <?JavaScript function funct1 (y){ ?>
       <?JavaScript var x;?>
       <?JavaScript x = y * 10;?>
       <?JavaScript return (x);?>
       <?JavaScript }?>
  </doc>
```

What is DTD?

- DTD defines the rules that set out how the document should be structured, what elements should be included, what kind of data may be included and what default values to use.
- Multiple documents and applications can share DTDs.
- DTDs use a formal grammar to describe the structure and syntax of an XML document, including the permissible values for much of that document's content.
- DTDs:

1. provide a formal and complete definition of an XML vocabulary.
2. are shareable descriptions of the structure of an XML documents.
3. are a way to validate specific instances of XML documents and constraints.
4. are restricted to one DTD per document instance.
5. specifies the validity of each tag.

Internal Vs. External DTD

- DTD may be divided into two parts: the internal subset and the external subset.
- These subsets are relative to the document instance.

    - The internal subset is a portion of the DTD including within the document.

    - The external subset is a portion of declarations that are located in a separate document.

- A DTD might be contained entirely within the document, with no external subset, or a document may simply refer to an external subset and contain no DTD declaration of its own.
- In many cases, DTD may use a combination of both.
- DTD declarations in the internal subset have priority over those in the external subset.

Associating a DTD with an XML document

- Each XML document can be associated with one, and only one DTD using single DOCTYPE declaration.
- The limit of one DTD per document can be an unfortune restriction.
- DTDs are linked to XML documents using markup called the Document Type Definitions.
- This declaration is commonly referred to as “the DOCTYPE declaration” to differentiate it from a DTD.

The Document Type (DOCTYPE) Declarations:

- A document type declaration is placed in an XML document’s prolog to say what DTD that document adheres to.
- It also specifies which element is the root element of the document.
- A document type *declaration* is not the same thing as a document type *definition*.
- A document type declaration must contain or refer to a document type definition, but a document type definition never contains a document type declaration.
- A document type declaration begins with <!DOCTYPE and ends with a >.
- A document type declaration has this basic form:

  <!DOCTYPE *name_of_root_element*
  SYSTEM “*URL of the external DTD subset*” [
  internal DTD subset
  ]>

- Here *name_of_root_element* is simply the name of the root element.
- The SYSTEM keyword indicates that what follows is a URL where the DTD is located.
- The square brackets enclose the internal subset of the DTD—that is, those declarations included inside the document itself.
- <u>The DOCTYPE declaration consists of:</u>

1. The usual XML tag delimiters (“<” and “?”).
2. The exclamation mark (“!”) that signifies a special XML declaration.
3. The DOCTYPE keyword.
4. The name of the document element (document_element).
5. One of two legal source keywords.

6. One of two DTD locations to associate an external DTD subset within a document.
7. Some additional declarations referring to the internal subset of the DTD.

Validating Against a DTD

- To be considered *valid,* an XML document must satisfy four criteria:

   1. It must be well formed.

   2. It must have a document type declaration.

   3. Its root element must be the one specified by the document type declaration.

   4. It must satisfy all the constraints indicated by the DTD specified by the document type declaration.

The Document Element Name

- The first variable of any DOCTYPE declaration is the name of the document element.
- This is required to be the root element of XML document.
- **Example:**

<?XML version = "1.0"?>

<!DOCTYPE Employee………..>

<Employee>

</Employee>

Basic DTD Declarations

- DTD declarations are delimited with the usual XML tag delimiters ("<" and ">").
- Like DOCTYPE declarations, all DTD declarations are indicated by the use of the

exclamation mark ("!") followed by a keyword, and its specific parameters

<! Keyword parameter1, paramenter2, …………, parameterN>

- There are four basic keywords used in DTD declarations

1. ELEMENT
2. ATTLIST
3. ENTITY
4. NOTATION

Element Type (ELEMENT) Declarations:

- Elements are described using the element type declaration.
- This declaration can have one of two different forms depending on the value of the category parameter

<!ELEMENT name Category>

<!ELEMENT name (Content_Model)>

Element content Categories

- There are 5 categories of element content:

Content Category	**Description**
ANY	Element type may contain any well formed XML data.
EMPTY	Element type may contain any text or child elements- only elements attributes are permitted.
Element	Element type contains only child elements no additional text is permitted.
Mixed	Element type may contain text and/or child element.
PCDATA	Element type may contain text (character data) only.

Content Models

- Content models are used to describe the structure and content of a given element type.
- The content may be:

1. Character data (PCDATA content).
2. One or more child element types (element-only content).
3. A combination of the two (mixed content).

- The key difference between element content and mixed content is the use of the **#PCDATA** keyword.
- If present, the content model is either mixed or PCDATA.
- The absence of this keyword indicates element-only content.

Cardinality

- Cardinality operators define how many child elements may appear in a content model.
- There are four cardinality operators:

Operators	Description
[none]	The absence of a cardinality operator character indicates that one, and only one, instance of child element is allowed (required).
?	Zero or one element – optional singular element.
*****	Zero or more element – optional element(s).
+	One or more child elements – required element(s).

/************* Example of Cardinality Operators ***************/

```
<!ELEMENT PersonName(
  (Mr./Miss) ?, First_Name +, Middle_Name *, Last_Name)>
```

The Attribute (ATTLIST) Declarations

- Attributes can be used to describe the meta-data or properties of the associated element.
- Element attributes are described using the attribute list declarations, also called ATTLIST declarations.
- This declaration has the usual DTD declarations format, using the ATTLIST keyword plus zero or more attribute definitions.

<! ATTLIST element_name attrName attrType attrDefault defaultValue>

/************* Example of Attribute List Declarations **************/

```
<!ELEMENT Book EMPTY>
<!ATTLIST Book
   isbn     CDATA  #REQUIRED
   title    CDATA  #REQUIRED
   author  CDATA  #REQUIRED
   price   CDATA  #IMPLIED (optional)
>
```

Attribute Types

• There are 10 different types of attributes defined in XML 1.0 recommendation.

Attribute	**Description**
CDATA	Character Data (simple text string)
Enumerated values (Choice list)	Attribute must be one of a series that is explicitly defined in DTD.
ID	Attribute value is the unique identifier for this element instance
IDREF	A reference to the element with an ID attribute that has the same value as that of IDREF
IDREFS	A list of IDREFs delimited by white space
NMTOKEN	A name token – a text string that confirms to the XML name rules
NMTOKENS	A list of NMTOKENs delimited by white spaces
ENTITY	The name of a pre-defined entity
ENTITIES	A list of ENTITY name delimited by white spaces
NOTATION	Attribute value must be a notation type that is explicitly declared elsewhere in the DTD

/************************** **Example of DTD** ***************************/

```
<!DOCTYPE Employees[
<!ELEMENT Employees (employee +)>
<!ELEMENT employee (Name +, Position +, Address +)>
<!ATTLIST employee id ID #REQUIRED>
<!ELEMENT Name (First +, Middle ?, Last +)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Middle(#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Position (#PCDATA)>
<!ELEMENT Address (Street ?, City +, State +, Zip *)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Zip (#PCDATA)>
]>
```

Limitations of DTD

- Some limitations of DTD include:
  1. DTD are not extensible, unlike XML itself.
  2. Only one DTD may be associated with each XML document.
  3. DTDs do not work well with XML namespaces.
  4. Supports very weak data typing.
  5. Limited content model descriptions.
  6. No object oriented type object inheritance.
  7. A document can override / ignore an external DTD using internal subset.
  8. Non-XML syntax.
  9. No DOM support.
  10. Relatively few older, more expensive tools.
  11. Very limited support to modularity and reuse.
  12. Too simple ID attribute mechanism (no points to requirements, uniqueness scope, etc)

DTDs

The purpose of a DTD (Document Type Definition) is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements and attributes.

Introduction to DTD

A Document Type Definition (DTD) defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements and attributes.

A DTD can be declared inline inside an XML document, or as an external reference.

Internal DTD Declaration

If the DTD is declared inside the XML file, it should be wrapped in a DOCTYPE definition with the following syntax:<!DOCTYPE root-element [element-declarations]>

Example XML document with an internal DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>

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

Open the XML file above in your browser and select view source or view page source to view the DTD. The DTD above is interpreted like this:

!DOCTYPE note defines that the root element of this document is note.

!ELEMENT note defines that the note element contains four elements: "to,from,heading,body".

!ELEMENT to defines the to element to be of the type "#PCDATA".

!ELEMENT from defines the from element to be of the type "#PCDATA".
!ELEMENT heading defines the heading element to be of the type "#PCDATA".
!ELEMENT body defines the body element to be of the type "#PCDATA".

External DTD Declaration

If the DTD is declared in an external file, it should be wrapped in a DOCTYPE definition with the following syntax :

```
<!DOCTYPE root-element SYSTEM "filename">
```

This is the same XML document as above, but with an external DTD (Open it, and select view source):

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

And this is the file "note.dtd" which contains the DTD:

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

Why Use a DTD?

- With a DTD, each of your XML files can carry a description of its own format.
- With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.
- Your application can use a standard DTD to verify that the data you receive from

the outside world is valid.

- You can also use a DTD to verify your own data.

DTD - XML Building Blocks

The main building blocks of both XML and HTML documents are elements.

The Building Blocks of XML Documents

Seen from a DTD point of view, all XML documents (and HTML documents) are made up by the following building blocks:

- Elements
- Attributes
- Entities
- PCDATA
- CDATA

Elements are the main building blocks of both XML and HTML documents.

Examples of HTML elements are "body" and "table". Examples of XML elements could be "note" and "message". Elements can contain text, other elements, or be empty. Examples of empty HTML elements are "hr", "br" and "img".

Examples:

```
<body>some text</body>
<message>some text</message>
```

Attributes

- Attributes provide extra information about elements.
- Attributes are always placed inside the opening tag of an element. Attributes always come in name/value pairs. The following "img" element has additional information about a source file:<img src="computer.gif" />
- The name of the element is "img". The name of the attribute is "src". The value of the attribute is "computer.gif". Since the element itself is empty it is closed by a " /".

Entities

- Some characters have a special meaning in XML, like the less than sign (<) that defines the start of an XML tag.
- Most of you know the HTML entity: "&nbsp;". This "no-breaking-space" entity is used in HTML to insert an extra space in a document. Entities are expanded when a document is parsed by an XML parser.
- The following entities are predefined in XML:

Entity References	Character
&lt;	<
&gt;	>
&amp;	&
&quot;	"
&apos;	'

PCDATA

- PCDATA means parsed character data.
- Think of character data as the text found between the start tag and the end tag of an XML element.
- PCDATA is text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.
- Tags inside the text will be treated as markup and entities will be expanded.
- However, parsed character data should not contain any &, <, or > characters; these

need to be represented by the &amp; &lt; and &gt; entities, respectively.

CDATA

- CDATA means character data.
- CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

DTD – Elements

In a DTD, elements are declared with an ELEMENT declaration.

Declaring Elements

In a DTD, XML elements are declared with an element declaration with the following syntax:

```
<!ELEMENT element-name category>
or
<!ELEMENT element-name (element-content)>
```

Empty Elements

Empty elements are declared with the category keyword EMPTY:

```
<!ELEMENT element-name EMPTY>
```

Example:

```
<!ELEMENT br EMPTY>
```

XML example:

```
<br />
```

Elements with Parsed Character Data

Elements with only parsed character data are declared with #PCDATA inside parentheses:

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

Example:

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

Elements with any Contents

Elements declared with the category keyword ANY, can contain any combination of parsable data:

<!ELEMENT element-name ANY>

Example:

<!ELEMENT note ANY>

Elements with Children (sequences)

Elements with one or more children are declared with the name of the children elements inside parentheses:

<!ELEMENT element-name (child1)>

or

<!ELEMENT element-name (child1,child2,...)>

Example:

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

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the "note" element is:

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

Declaring Only One Occurrence of an Element:

```
        <!ELEMENT element-name (child-name)>
```

Example:

```
        <!ELEMENT note (message)>
```

The example above declares that the child element "message" must occur once, and only once inside the "note" element.

Declaring Minimum One Occurrence of an Element

<!ELEMENT element-name (child-name+)>
Example:

    <!ELEMENT note (message+)>

The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.
Declaring Zero or More Occurrences of an Element <!ELEMENT element-name (child-name*)>
Example:

    <!ELEMENT note (message*)>

The * sign in the example above declares that the child element "message" can occur zero or more times inside the "note" element.
Declaring Zero or One Occurrences of an Element <!ELEMENT element-name (child-name?)>
Example:

    <!ELEMENT note (message?)>

The ? sign in the example above declares that the child element "message" can occur zero or one time inside the "note" element.
Declaring either/or ContentExample:
<!ELEMENT note (to,from,header,(message|body))>

The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.
Declaring Mixed ContentExample:
<!ELEMENT note (#PCDATA|to|from|header|message)*>

The example above declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

DTD – Attributes

In a DTD, attributes are declared with an ATTLIST declaration.

Declaring Attributes

An attribute declaration has the following syntax:

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

DTD example:

```
<!ATTLIST payment type CDATA "check">
```

XML example:

```
<payment type="check" />
```

The attribute-type can be one of the following:

CDATA	-	The value is character data
(en1|en2|..)	-	The value must be one from an enumerated list
ID	-	The value is a unique id
IDREF	-	The value is the id of another element
IDREFS	-	The value is a list of other ids
NMTOKEN	-	The value is a valid XML name
NMTOKENS	-	The value is a list of valid XML names
ENTITY	-	The value is an entity
ENTITIES	-	The value is a list of entities
NOTATION	-	The value is a name of a notation

XML:

The value is a predefined xml value. The default-value can be one of the following:

Value	-	The default value of the attribute
#REQUIRED	-	The attribute is required
#IMPLIED	-	The attribute is not required
#FIXED value	-	The attribute value is fixed

A Default Attribute ValueDTD:

```
<!ELEMENT square EMPTY>
<!ATTLIST square width CDATA "0">
```

Valid XML:

```
<square width="100" />
```

In the example above, the "square" element is defined to be an empty element with a "width" attribute of type CDATA. If no width is specified, it has a default value of 0.

#REQUIRED

Syntax:

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

Example DTD:

```
<!ATTLIST person number CDATA #REQUIRED>
```

Valid XML:

```
<person number="5677" />
```

Invalid XML:

```
<person />
```

Use the #REQUIRED keyword if you don't have an option for a default value, but still want to force the attribute to be present.

#IMPLIED

Syntax:

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

ExampleDTD:

```
<!ATTLIST contact fax CDATA #IMPLIED>
```

Valid XML:

```
<contact fax="555-667788" />
```

Valid XML:

```
<contact />
```

Use the #IMPLIED keyword if you don't want to force the author to include an attribute, and you don't have an option for a default value.

#FIXED

Syntax

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

ExampleDTD:

```
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

Valid XML:

```
<sender company="Microsoft" />
```

Invalid XML:

```
<sender company="W3Schools"     />
```

Use the #FIXED keyword when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

Enumerated Attribute Values

Syntax

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

ExampleDTD:

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

Use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values.

XML Elements vs. Attributes

In XML, there are no rules about when to use attributes, and when to use child elements.

Use of Elements vs. Attributes

Data can be stored in child elements or in attributes.

Take a look at these examples:

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

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

There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

My Favorite Way

I like to store data in child elements.

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="12/11/2002">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

A date element is used in the second example:

```
<note>
<date>12/11/2002</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

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

```
<note>
<date>
  <day>12</day>
  <month>11</month>
  <year>2002</year>
</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
```

```
	<body>Don't forget me this weekend!</body>
	</note>
```

Avoid using attributes?

Should you avoid using attributes?

Some of the problems with attributes are:

- attributes cannot contain multiple values (child elements can)
- attributes are not easily expandable (for future changes)
- attributes cannot describe structures (child elements can)
- attributes are more difficult to manipulate by program code
- attribute values are not easy to test against a DTD

If you use attributes as containers for data, you end up with documents that are difficult to read and maintain. Try to use elements to describe data. Use attributes only to provide information that is not relevant to the data.

Don't end up like this (this is not how XML should be used):

```
<note day="12" month="11" year="2002"
to="Tove" from="Jani" heading="Reminder"
body="Don't forget me this weekend!">
</note>
```

An Exception to my Attribute Rule

Rules always have exceptions. My rule about attributes has one exception:
Sometimes I assign ID references to elements. These ID references can be used to access XML elements in much the same way as the NAME or ID attributes in HTML. This example demonstrates this:

```
<messages>
  <note id="p501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="p502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not!</body>
  </note>
</messages>
```

The ID in these examples is just a counter, or a unique identifier, to identify the different notes in the XML file, and not a part of the note data.
What I am trying to say here is that metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

DTD – Entities

Entities are variables used to define shortcuts to standard text or special characters.

Entity references are references to entities

Entities can be declared internal or external

An Internal Entity Declaration

Syntax

```
<!ENTITY entity-name "entity-value">
```

Example DTD:

```
<!ENTITY writer "Donald Duck.">
<!ENTITY copyright "Copyright W3Schools.">
```

XML example:

```
<author>&writer;&copyright;</author>
```

Note: An entity has three parts: an ampersand (&), an entity name, and a semicolon (;).

An External Entity Declaration

Syntax<!ENTITY entity-name SYSTEM "URI/URL">

ExampleDTD Example:

```
<!ENTITY writer SYSTEM "http://www.w3schools.com/entities.dtd">
<!ENTITY copyright SYSTEM "http://www.w3schools.com/entities.dtd">
```

XML example:

```
<author>&writer;&copyright;</author>
```

DTD Validation

With Internet Explorer 5+ you can validate your XML against a DTD.

Validating With the XML Parser

If you try to open an XML document, the XML Parser might generate an error. By accessing the parseError object, you can retrieve the error code, the error text, or even the line that caused the error.

Note: The load( ) method is used for files, while the loadXML( ) method is used for strings.var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");

```
xmlDoc.async="false";
xmlDoc.validateOnParse="true";
xmlDoc.load("note_dtd_error.xml");
document.write("<br />Error Code: ");
document.write(xmlDoc.parseError.errorCode);
document.write("<br />Error Reason: ");
document.write(xmlDoc.parseError.reason);
document.write("<br />Error Line: ");
document.write(xmlDoc.parseError.line);
```

Turning Validation Off

Validation can be turned off by setting the XML parser's validateOnParse="false". var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");

```
xmlDoc.async="false";
xmlDoc.validateOnParse="false";
xmlDoc.load("note_dtd_error.xml");
document.write("<br />Error Code: ");
document.write(xmlDoc.parseError.errorCode);
document.write("<br />Error Reason: ");
document.write(xmlDoc.parseError.reason);
document.write("<br />Error Line: ");
document.write(xmlDoc.parseError.line);
```

DTD - Examples from the Net

TV Schedule DTD

```
    <!DOCTYPE TVSCHEDULE [
    <!ELEMENT TVSCHEDULE (CHANNEL+)>
    <!ELEMENT CHANNEL (BANNER,DAY+)>
    <!ELEMENT BANNER (#PCDATA)>
    <!ELEMENT DAY (DATE,(HOLIDAY|PROGRAMSLOT+)+)>
    <!ELEMENT HOLIDAY (#PCDATA)>
    <!ELEMENT DATE (#PCDATA)>
    <!ELEMENT PROGRAMSLOT (TIME,TITLE,DESCRIPTION?)>
    <!ELEMENT TIME (#PCDATA)>
```

```
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ATTLIST TVSCHEDULE NAME CDATA #REQUIRED>
<!ATTLIST CHANNEL CHAN CDATA #REQUIRED>
<!ATTLIST PROGRAMSLOT VTR CDATA #IMPLIED>
<!ATTLIST TITLE RATING CDATA #IMPLIED>
<!ATTLIST TITLE LANGUAGE CDATA #IMPLIED>
]>
```

Newspaper Article DTD

```
 <!DOCTYPE NEWSPAPER [
<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>
<!ENTITY NEWSPAPER "Vervet Logic Times">
<!ENTITY PUBLISHER "Vervet Logic Press">
<!ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">
]>
```

Product Catalog DTD

```
<!DOCTYPE CATALOG [
<!ENTITY AUTHOR "John Doe">
<!ENTITY COMPANY "JD Power Tools, Inc.">
<!ENTITY EMAIL "jd@jd-tools.com">
<!ELEMENT CATALOG (PRODUCT+)>
<!ELEMENT PRODUCT
(SPECIFICATIONS+,OPTIONS?,PRICE+,NOTES?)>
<!ATTLIST PRODUCT
NAME CDATA #IMPLIED
CATEGORY (HandTool|Table|Shop-Professional) "HandTool"
PARTNUM CDATA #IMPLIED
PLANT (Pittsburgh|Milwaukee|Chicago) "Chicago"
INVENTORY (InStock|Backordered|Discontinued) "InStock">

<!ELEMENT SPECIFICATIONS (#PCDATA)>
<!ATTLIST SPECIFICATIONS
WEIGHT CDATA #IMPLIED
POWER CDATA #IMPLIED>
<!ELEMENT OPTIONS (#PCDATA)>
<!ATTLIST OPTIONS
FINISH (Metal|Polished|Matte) "Matte"
ADAPTER (Included|Optional|NotApplicable) "Included"
CASE (HardShell|Soft|NotApplicable) "HardShell">
<!ELEMENT PRICE (#PCDATA)>
<!ATTLIST PRICE
MSRP CDATA #IMPLIED
WHOLESALE CDATA #IMPLIED
STREET CDATA #IMPLIED
SHIPPING CDATA #IMPLIED>
<!ELEMENT NOTES (#PCDATA)>
]>
```

DTD Summary

This tutorial has taught you how to describe the structure of an XML document. You have learned how to use a DTD to define the legal elements of an XML document, and how a DTD can be declared inside your XML document, or as an external reference. You have learned how to declare the legal elements, attributes, entities, and CDATA sections for XML documents.

Now You Know DTD, What's Next?

The next step is to learn about XML Schema. XML Schema is used to define the legal elements of an XML document, just like a DTD. We think that very soon XML Schemas will be used in most Web applications as a replacement for DTDs. XML Schema is an XML-based alternative to DTD. Unlike DTD, XML Schemas has support for data types and namespaces.

Let’s learn XML Schema.

XML Schema

- XML Schema is an XML-based alternative to DTDs.
- An XML Schema describes the structure of an XML document.
- The XML Schema language is also referred to as XML Schema Definition (XSD).

Learn how to read and create XML Schemas, why XML Schemas are more powerful than DTDs, and how to use the XML Schema language in your application.

Introduction to XML Schema

- XML Schema is an XML-based alternative to DTD.
- An XML schema describes the structure of an XML document.
- The XML Schema language is also referred to as XML Schema Definition (XSD).

What You Should Already Know?

Before you continue you should have a basic understanding of the following:

- HTML / XHTML
- XML and XML Namespaces
- A basic understanding of DTD

What is an XML Schema?

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

An XML Schema defines:

- elements that can appear in a document
- attributes that can appear in a document
- which elements are child elements

- the order of child elements
- the number of child elements
- whether an element is empty or can include text
- data types for elements and attributes
- default and fixed values for elements and attributes

'XML Schemas are the Successors of DTDs'

We think that very soon XML Schemas will be used in most Web applications as a replacement for DTDs. Here are some reasons:

- XML Schemas are extensible to future additions
- XML Schemas are richer and more powerful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types
- XML Schemas support namespaces
- XML Schema is a W3C Standard

'XML Schema became a W3C Recommendation 02. May 2001.'

'XML Schemas are much more powerful than DTDs.'

XML Schemas Support Data Types

One of the greatest strength of XML Schemas is the support for data types.

With support for data types:

- It is easier to describe allowable document content
- It is easier to validate the correctness of data
- It is easier to work with data from a database
- It is easier to define data facets (restrictions on data)
- It is easier to define data patterns (data formats)
- It is easier to convert data between different data types

'XML Schemas use XML Syntax'.

Another great strength about XML Schemas is that they are written in XML.

Some benefits of that XML Schema are written in XML:

- You don't have to learn a new language
- You can use your XML editor to edit your Schema files
- You can use your XML parser to parse your Schema files
- You can manipulate your Schema with the XML DOM
- You can transform your Schema with XSLT
- XML Schemas Secure Data Communication

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

With XML Schemas, the sender can describe the data in a way that the receiver will understand.

A date like: "03-11-2004" will, in some countries, be interpreted as 3.November and in other countries as 11.March.

However, XML elements with a data type like this:

<date type="date">2004-03-11</date>

Ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

'XML Schemas are extensible, because they are written in XML.'

With an extensible Schema definition you can:

- Reuse your Schema in other Schemas
- Create your own data types derived from the standard types
- Reference multiple schemas in the same document

Well-Formed is not enough

A well-formed XML document is a document that conforms to the XML syntax rules, like:

- it must begin with the XML declaration
- it must have one unique root element

- start-tags must have matching end-tags
- elements are case sensitive
- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted
- entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.
Think of the following situation: you order 5 gross of laser printers, instead of 5 laser printers. With XML Schemas, most of these errors can be caught by your validating software.

A Simple XML Document

Look at this simple XML document called "note.xml":<?xml version="1.0"?>

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

A DTD File

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

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

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

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

An XML Schema

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

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

The note element is a complex type because it contains other elements. The other elements (to, from, heading, body) are simple types because they do not contain other elements. You will learn more about simple and complex types in the following chapters.

A Reference to a DTD

This XML document has a reference to a DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "http://www.w3schools.com/dtd/note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't    forget    me    this  weekend!</body>
```

```
</note>
```

A Reference to an XML Schema

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note
xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

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

The <schema> Element

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

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

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

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

The following

fragment:xmlns:xs="http://www.w3.org/2001/XMLSchema"

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

This fragment:targetNamespace="http://www.w3schools.com"

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

This fragment:xmlns="http://www.w3schools.com"

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

This fragment:elementFormDefault="qualified"

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

Referencing a Schema in an XML Document

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The following fragment:xmlns="http://www.w3schools.com"

Specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.w3schools.com" namespace.

Once you have the XML Schema Instance namespace

available:xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
You can use the schemaLocation attribute. This attribute has two values. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace:xsi:schemaLocation=http://www.w3schools.com note.xsd

XML Schemas define the elements of your XML files.

A simple element is an XML element that contains only text. It cannot contain any other elements or attributes.

What is a Simple Element?

A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes.
However, the "only text" restriction is quite misleading. The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself.
You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

Defining a Simple Element

The syntax for defining a simple element is: <xs:element name="xxx" type="yyy"/>
Where xxx is the name of the element and yyy is the data type of the element.
XML Schema has a lot of built-in data types. The most common types are:

```
xs:string
xs:decimal
xs:integer
xs:boolean
xs:date
xs:time
```

Example:

Here are some XML elements:

```
<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

And

Here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements

Simple elements may have a default value OR a fixed value specified.

A default value is automatically assigned to the element when no other value is specified.

In the following example the default value is "red":<xs:element name="color" type="xs:string" default="red"/>

A fixed value is also automatically assigned to the element, and you cannot specify another value.

In the following example the fixed value is "red":<xs:element name="color" type="xs:string" fixed="red"/>

'All attributes are declared as simple types.'

What is an Attribute?

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type.

How to Define an Attribute?

The syntax for defining an attribute is:

```
<xs:attribute name="xxx" type="yyy"/>
```

Where xxx is the name of the attribute and yyy specifies the data type of the attribute.

XML Schema has a lot of built-in data types. The most common types are:

```
xs:string
xs:decimal
xs:integer
xs:boolean
xs:date
xs:time
```

Example

Here is an XML element with an attribute:<lastname lang="EN">Smith</lastname> And here is the corresponding attribute definition:<xs:attribute name="lang" type="xs:string"/>

Default and Fixed Values for Attributes

Attributes may have a default value OR a fixed value specified.

A default value is automatically assigned to the attribute when no other value is specified.

In the following example the default value is "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

A fixed value is also automatically assigned to the attribute, and you cannot specify another value. In the following example the fixed value is "EN":<xs:attribute name="lang" type="xs:string" fixed="EN"/>

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:<xs:attribute name="lang" type="xs:string" use="required"/>

Restrictions on Content

When an XML element or attribute has a data type defined, it puts restrictions on the element's or attribute's content.

If an XML element is of type "xs:date" and contains a string like "Hello World", the element will not validate.

With XML Schemas, you can also add your own restrictions to your XML elements and attributes. These restrictions are called facets. Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.

Restrictions on Values

The following example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="120"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions on a Set of Values

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint.

The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The example above could also have been written like this:

```
<xs:element name="car" type="carType"/>
<xs:simpleType name="carType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
```

'In this case the type "carType" can be used by other elements because it is not a part of the "car" element.'

Restrictions on a Series of Values

To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint.

The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "initials" with a restriction. The only acceptable value is THREE of the UPPERCASE letters from a to z:

```
<xs:element name="initials">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z][A-Z][A-Z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example also defines an element called "initials" with a restriction. The only acceptable value is THREE of the LOWERCASE OR UPPERCASE letters from a to z:

```
<xs:element name="initials">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "choice" with a restriction. The only acceptable value is ONE of the following letters: x, y, OR z:

```
<xs:element name="choice">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[xyz]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "prodid" with a restriction. The only acceptable value is FIVE digits in a sequence, and each digit must be in a range from 0 to 9:

```
<xs:element name="prodid">
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:pattern      value="[0-9][0-9][0-9][0-   9][0-9]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Other Restrictions on a Series of Values

The example below defines an element called "letter" with a restriction. The acceptable value is zero or more occurrences of lowercase letters from a to z:

```
<xs:element name="letter">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="([a-z])*"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example also defines an element called "letter" with a restriction. The acceptable value is one or more pairs of letters, each pair consisting of a lower case letter followed by an upper case letter. For example, "sToP" will be validated by this pattern, but not "Stop" or "STOP" or "stop":

```
<xs:element name="letter">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="([a-z][A-Z])+"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "gender" with a restriction. The only acceptable value is male OR female:

```
<xs:element name="gender">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="male|female"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "password" with a restriction. There must be exactly eight characters in a row and those characters must be lowercase or uppercase letters from a to z, or a number from 0 to 9:

```
<xs:element name="password">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z0-9]{8}"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions on Whitespace Characters

To specify how whitespace characters should be handled, we would use the whiteSpace constraint.

This example defines an element called "address" with a restriction. The whiteSpace constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="preserve"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "replace", which means that the XML processor WILL REPLACE all white space characters (line feeds, tabs, spaces, and carriage returns) with spaces:

```
<xs:element name="address">
<xs:simpleType>
```

```
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="replace"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "collapse", which means that the XML processor WILL REMOVE all white space characters (line feeds, tabs, spaces, carriage returns are replaced with spaces, leading and trailing spaces are removed, and multiple spaces are reduced to a single space):

```
<xs:element name="address">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints.

This example defines an element called "password" with a restriction. The value must be exactly eight characters:

```
<xs:element name="password">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:length value="8"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

```
<xs:element name="password">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:minLength value="5"/>
    <xs:maxLength value="8"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions for Data types

Enumeration - *Defines a list of acceptable values.*

fractionDigits - *Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero.*

length *- Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero.*

maxExclusive - *Specifies the upper bounds for numeric values (the value must be less than this value).*

maxInclusive - *Specifies the upper bounds for numeric values (the value must be less than or equal to this value).*

maxLength* - *Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero.

minExclusive* - *Specifies the lower bounds for numeric values (the value must be greater than this value).

minInclusive* - *Specifies the lower bounds for numeric values (the value must be greater than or equal to this value).

minLength - *Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero.*

pattern - *Defines the exact sequence of characters that are acceptable.*

totalDigits* - *Specifies the exact number of digits allowed. Must be greater than zero.

whitespace* - *Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled.

XSD Complex Elements

A complex element contains other elements and/or attributes.

What is a Complex Element?

A complex element is an XML element that contains other elements and/or attributes.

There are four kinds of complex elements:

1. empty elements
2. elements that contain only other elements
3. elements that contain only text
4. elements that contain both other elements and text

'Each of these elements may contain attributes as well!'

Examples of Complex Elements

A complex XML element, "product", which is empty:

<product pid="1345"/>

A complex XML element, "employee", which contains only other elements:

<employee>

<firstname>John</firstname>

```
        <lastname>Smith</lastname>
    </employee>
```

A complex XML element, "food", which contains only text:

```
    <food type="dessert">Ice cream</food>
```

A complex XML element, "description", which contains both elements and text:

```
    <description> It happened on <date lang="norwegian">03.03.99</date>
    ....</description>
```

How to Define a Complex Element

Look at this complex XML element, "employee", which contains only other elements:

```
<employee>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
</employee>
```

We can define a complex element in an XML Schema two different ways:

1. The "employee" element can be declared directly by naming the element, like this:

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

If you use the method described above, only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared. You will learn more about indicators in the XSD Indicators chapter.

2. The "employee" element can have a type attribute that refers to the name of the

complex type to use:

```
<xs:element name="employee" type="personinfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

If you use the method described above, several elements can refer to the same complex type, like this:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

You can also base a complex element on an existing complex element and add some elements, like this:

```
<xs:element name="employee" type="fullpersoninfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
    <xs:element name="address" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

XSD Complex Empty Elements

An empty complex element cannot have contents, only attributes.

Complex Empty Elements

An empty XML element:<product prodid="1345" />

The "product" element above has no content at all. To define a type with no content, we must define a type that allows only elements in its content, but we do not actually declare any elements, like this:

```
<xs:element name="product">
 <xs:complexType>
  <xs:complexContent>
   <xs:restriction base="xs:integer">
    <xs:attribute name="prodid" type="xs:positiveInteger"/>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
```

In the example above, we define a complex type with a complex content. The complexContent element signals that we intend to restrict or extend the content model of a complex type, and the restriction of integer declares one attribute but does not introduce any element content.

However, it is possible to declare the "product" element more compactly, like this:

```
<xs:element name="product">
 <xs:complexType>
  <xs:attribute             name="prodid"  type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
```

Or

you can give the complexType element a name, and let the "product" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="product" type="prodtype"/>
<xs:complexType name="prodtype">
  <xs:attribute name="prodid" type="xs:positiveInteger"/>
</xs:complexType>
```

XSD Complex Type - Elements Only

An "elements-only" complex type contains an element that contains only other elements. Complex Types Containing Elements Only. An XML element, "person", that contains only other elements:

```
<person>
<firstname>John</firstname>
<lastname>Smith</lastname>
</person>
```

You can define the "person" element in a schema, like this:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Notice the <xs:sequence> tag. It means that the elements defined ("firstname" and "lastname") must appear in that order inside a "person" element.

Or

You can give the complexType element a name, and let the "person" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="person" type="persontype"/>
<xs:complexType name="persontype">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

XSD Complex Text-Only Elements

A complex text-only element can contain text and attributes.

Complex Text-Only Elements

This type contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simple content, you must define an extension OR a restriction within the simpleContent element, like this:

```
<xs:element name="somename">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="basetype">
        ....
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

OR

```
<xs:element name="somename">
  <xs:complexType>
    <xs:simpleContent>
      <xs:restriction base="basetype">
        ....
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

'Use the extension/restriction element to expand or to limit the base simple type for the element.'

Here is an example of an XML element, "shoesize", that contains text-only:

```
    <shoesize country="france">35</shoesize>
```

The following example declares a complexType, "shoesize". The content is defined as an integer value, and the "shoesize" element also contains an attribute named "country":

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

We could also give the complexType element a name, and let the "shoesize" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="shoesize" type="shoetype"/>
<xs:complexType name="shoetype">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="country" type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

XSD Complex Types with Mixed Content

A mixed complex type element can contain attributes, elements, and text.

Complex Types with Mixed Content

An XML element, "letter", that contains both text and other elements:

```
<letter>
      Dear Mr.<name>John Smith</name>.
      Your order <orderid>1032</orderid>
      will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

The following schema declares the "letter" element:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note: To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true". The <xs:sequence> tag means that the elements defined (name, orderid and shipdate) must appear in that order inside a "letter" element.

We could also give the complexType element a name, and let the "letter" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="letter" type="lettertype"/>
<xs:complexType name="lettertype" mixed="true">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="orderid" type="xs:positiveInteger"/>
    <xs:element name="shipdate" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
```

XSD Complex Types Indicators

We can control HOW elements are to be used in documents with indicators.

Indicators

There are seven indicators:

1. Order indicators:
2. All
3. Choice
4. Sequence
5. Occurrence indicators:
    a. maxOccurs
    b. minOccurs
6. Group indicators:
    a. Group name
    b. attributeGroup name

7. Order Indicators

Order indicators are used to define the order of the elements.

All Indicators

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Note: When using the <all> indicator you can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1 (the <minOccurs> and <maxOccurs> are described later).

Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employee"/>
      <xs:element name="member" type="member"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

```
<xs:element name="person">
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

Note: For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) the default value for maxOccurs and minOccurs is 1.

maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of one time (the default value for minOccurs is 1) and a maximum of ten times in the "person" element.

minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:<xs:element name="person">

```
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of zero times and a maximum of ten times in the "person" element.

Tip: To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement:

A working example: An XML file called "Myfamily.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="family.xsd">
<person>
<full_name>Hege Refsnes</full_name>
<child_name>Cecilie</child_name>
</person>
<person>
<full_name>Tove Refsnes</full_name>
<child_name>Hege</child_name>
<child_name>Stale</child_name>
<child_name>Jim</child_name>
<child_name>Borge</child_name>
</person>
```

```
<person>
<full_name>Stale Refsnes</full_name>
</person>
</persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full_name" element and it can contain up to five "child_name" elements.

Here is the schema file "family.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="persons">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="person" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
      minOccurs="0" maxOccurs="5"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Group Indicators

Group indicators are used to define related sets of elements.

Element Groups

Element groups are defined with the group declaration, like this:

```
<xs:group name="groupname">
 ...
</xs:group>
```

You must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup", that defines a group of elements that must occur in an exact sequence:

```
<xs:group name="persongroup">
 <xs:sequence>
  <xs:element name="firstname" type="xs:string"/>
  <xs:element name="lastname" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
 </xs:sequence>
</xs:group>
```

After you have defined a group, you can reference it in another definition, like this:

```
<xs:group name="persongroup">
 <xs:sequence>
  <xs:element name="firstname" type="xs:string"/>
  <xs:element name="lastname" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
 </xs:sequence>
</xs:group>
<xs:element name="person" type="personinfo"/>
<xs:complexType name="personinfo">
 <xs:sequence>
  <xs:group ref="persongroup"/>
  <xs:element name="country" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

Attribute Groups

Attribute groups are defined with the attributeGroup declaration, like this:

```
<xs:attributeGroup name="groupname">
 ...
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<xs:attributeGroup name="personattrgroup">
  <xs:attribute name="firstname" type="xs:string"/>
  <xs:attribute name="lastname" type="xs:string"/>
  <xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
```

After you have defined an attribute group, you can reference it in another definition, like this:

```
<xs:attributeGroup name="personattrgroup">
  <xs:attribute name="firstname" type="xs:string"/>
  <xs:attribute name="lastname" type="xs:string"/>
  <xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
<xs:element name="person">
  <xs:complexType>
    <xs:attributeGroup ref="personattrgroup"/>
  </xs:complexType>
</xs:element>
```

XSD The <any> Element

The <any> element enables us to extend the XML document with elements not specified by the schema!

The <any> Element

The <any> element enables us to extend the XML document with elements not specified by the schema.

The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <any> element we can extend (after <lastname>) the content of "person" with any element:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:any minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Now we want to extend the "person" element with a "children" element. In this case we can do so, even if the author of the schema above never declared any "children" element.
Look at this schema file, called "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">
<xs:element name="children">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="childname" type="xs:string"
      maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persons                                   xmlns="http://www.microsoft.com"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://www.microsoft.com family.xsd
http://www.w3schools.com children.xsd">
<person>
<firstname>Hege</firstname>
<lastname>Refsnes</lastname>
<children>
  <childname>Cecilie</childname>
</children>
</person>
<person>
<firstname>Stale</firstname>
<lastname>Refsnes</lastname>
</person>
</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to extend the "person" element with an optional element after the "lastname" element.

The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSD The <anyAttribute> Element

The <anyAttribute> element enables us to extend the XML document with attributes not specified by the schema!

The <anyAttribute> Element

The <anyAttribute> element enables us to extend the XML document with attributes not specified by the schema.

The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <anyAttribute> element we can add any number of attributes to the "person" element:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
```

Now we want to extend the "person" element with a "gender" attribute. In this case we can do so, even if the author of the schema above never declared any "gender" attribute.

Look at this schema file, called "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">
<xs:attribute name="gender">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="male|female"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persons xmlns="http://www.microsoft.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://www.microsoft.com family.xsd
http://www.w3schools.com attribute.xsd">
<person gender="female">
<firstname>Hege</firstname>
<lastname>Refsnes</lastname>
</person>
<person gender="male">
<firstname>Stale</firstname>
<lastname>Refsnes</lastname>
</person>
</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to add an attribute to the "person" element. The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSD Element Substitution

With XML Schemas, one element can substitute another element.

Element Substitution

Let's say that we have users from two different countries: England and Norway. We would like the ability to let the user choose whether he or she would like to use the Norwegian element names or the English element names in the XML document.

To solve this problem, we could define a substitutionGroup in the XML schema. First, we declare a head element and then we declare the other elements which state that they are substitutable for the head element.<xs:element name="name" type="xs:string"/>

```
<xs:element name="navn" substitutionGroup="name"/>
```

In the example above, the "name" element is the head element and the "navn" element is substitutable for "name".

Look at this fragment of an XML schema:

```
<xs:element name="name" type="xs:string"/>
<xs:element name="navn" substitutionGroup="name"/>
<xs:complexType name="custinfo">
  <xs:sequence>
    <xs:element ref="name"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="customer" type="custinfo"/>
<xs:element name="kunde" substitutionGroup="customer"/>
```

A valid XML document (according to the schema above) could look like this:

```
<customer>
  <name>John Smith</name>
</customer>
```

Or

Like this:

```
<kunde>
  <navn>John Smith</navn>
</kunde>
```

Blocking Element Substitution

To prevent other elements from substituting with a specified element, use the block attribute:

```
        <xs:element name="name" type="xs:string" block="substitution"/>
```

Look at this fragment of an XML schema:

```
<xs:element name="name" type="xs:string" block="substitution"/>
<xs:element name="navn" substitutionGroup="name"/>
<xs:complexType name="custinfo">
  <xs:sequence>
    <xs:element ref="name"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="customer" type="custinfo" block="substitution"/>
<xs:element name="kunde" substitutionGroup="customer"/>
```

A valid XML document (according to the schema above) looks like this:

```
<customer>
  <name>John Smith</name>
</customer>
```

BUT THIS IS NO LONGER VALID:

```
<kunde>
  <navn>John Smith</navn>
</kunde>
```

Using substitutionGroup

The type of the substitutable elements must be the same as, or derived from, the type of the head element. If the type of the substitutable element is the same as the type of the head element you will not have to specify the type of the substitutable element.

Note that all elements in the substitutionGroup (the head element and the substitutable elements) must be declared as global elements, otherwise it will not work!

What are Global Elements?

Global elements are elements that are immediate children of the "schema" element! Local elements are elements nested within other elements.

An XSD Example

This chapter will demonstrate how to write an XML Schema. You will also learn that a schema can be written in different ways.

An XML Document

Let's have a look at this XML document called "shiporder.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
 <orderperson>John Smith</orderperson>
 <shipto>
  <name>Ola Nordmann</name>
  <address>Langgt 23</address>
  <city>4000 Stavanger</city>
  <country>Norway</country>
 </shipto>
 <item>
  <title>Empire Burlesque</title>
  <note>Special Edition</note>
  <quantity>1</quantity>
  <price>10.90</price>
 </item>
 <item>
  <title>Hide your heart</title>
  <quantity>1</quantity>
  <price>9.90</price>
 </item>
</shiporder>
```

The XML document above consists of a root element, "shiporder", that contains a required attribute called "orderid". The "shiporder" element contains three different child elements: "orderperson", "shipto" and "item". The "item" element appears twice, and it contains a

"title", an optional "note" element, a "quantity", and a "price" element.
The line above: xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" tells the XML parser that this document should be validated against a schema. The line: xsi:noNamespaceSchemaLocation="shiporder.xsd" specifies WHERE the schema resides (here it is in the same folder as "shiporder.xml").

Create an XML Schema

Now we want to create a schema for the XML document above.
We start by opening a new file that we will call "shiporder.xsd". To create the schema we could simply follow the structure in the XML document and define each element as we find it. We will start with the standard XML declaration followed by the xs:schema element that defines a schema:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
</xs:schema>
```

In the schema above we use the standard namespace (xs), and the URI associated with this namespace is the Schema language definition, which has the standard value of http://www.w3.org/2001/XMLSchema.

Next, we have to define the "shiporder" element. This element has an attribute and it contains other elements, therefore we consider it as a complex type. The child elements of the "shiporder" element is surrounded by a xs:sequence element that defines an ordered sequence of sub elements:<xs:element name="shiporder">

```
 <xs:complexType>
  <xs:sequence>
  ...
  </xs:sequence>
  ...
 </xs:complexType>
</xs:element>
```

Then we have to define the "orderperson" element as a simple type (because it does not contain any attributes or other elements). The type (xs:string) is prefixed with the namespace prefix associated with XML Schema that indicates a predefined schema data type:<xs:element name="orderperson" type="xs:string"/>

Next, we have to define two elements that are of the complex type: "shipto" and "item". We start by defining the "shipto" element:<xs:element name="shipto">

```
 <xs:complexType>
  <xs:sequence>
   <xs:element name="name" type="xs:string"/>
   <xs:element name="address" type="xs:string"/>
   <xs:element name="city" type="xs:string"/>
   <xs:element name="country" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

With schemas we can define the number of possible occurrences for an element with the maxOccurs and minOccurs attributes. maxOccurs specifies the maximum number of occurrences for an element and minOccurs specifies the minimum number of occurrences for an element. The default value for both maxOccurs and minOccurs is 1!

Now we can define the "item" element. This element can appear multiple times inside a "shiporder" element. This is specified by setting the maxOccurs attribute of the "item" element to "unbounded" which means that there can be as many occurrences of the "item" element as the author wishes. Notice that the "note" element is optional.

We have specified this by setting the minOccurs attribute to zero:

```
<xs:element name="item" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="title" type="xs:string"/>
   <xs:element name="note" type="xs:string" minOccurs="0"/>
   <xs:element name="quantity" type="xs:positiveInteger"/>
   <xs:element name="price" type="xs:decimal"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

We can now declare the attribute of the "shiporder" element. Since this is a required attribute we specify use="required".

Note: The attribute declarations must always come last:<xs:attribute name="orderid" type="xs:string" use="required"/>

Here is the complete listing of the schema file called "shiporder.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="orderperson" type="xs:string"/>
   <xs:element name="shipto">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="item" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string" minOccurs="0"/>
      <xs:element name="quantity" type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="orderid" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Divide the Schema

The previous design method is very simple, but can be difficult to read and maintain when documents are complex.

The next design method is based on defining all elements and attributes first, and then referring to them using the ref attribute.

Here is the new design of the schema file ("shiporder.xsd"):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- definition of simple elements -->
<xs:element name="orderperson" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="country" type="xs:string"/>
<xs:element name="title" type="xs:string"/>
<xs:element name="note" type="xs:string"/>
<xs:element                    name="quantity"   type="xs:positiveInteger"/>
```

```
<xs:element name="price" type="xs:decimal"/>

<!-- definition of attributes -->
<xs:attribute name="orderid" type="xs:string"/>

<!-- definition of complex elements -->
<xs:element name="shipto">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="name"/>
   <xs:element ref="address"/>
   <xs:element ref="city"/>
   <xs:element ref="country"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="item">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="title"/>
   <xs:element ref="note" minOccurs="0"/>
   <xs:element ref="quantity"/>
   <xs:element ref="price"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="shiporder">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="orderperson"/>
   <xs:element ref="shipto"/>
   <xs:element ref="item" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="orderid" use="required"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Using Named Types

The third design method defines classes or types, that enables us to reuse element definitions. This is done by naming the simpleTypes and complexTypes elements, and then point to them through the type attribute of the element.

Here is the third design of the schema file ("shiporder.xsd"):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:simpleType name="stringtype">
 <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="inttype">
 <xs:restriction base="xs:positiveInteger"/>
</xs:simpleType>
<xs:simpleType name="dectype">
 <xs:restriction base="xs:decimal"/>
</xs:simpleType>
<xs:simpleType name="orderidtype">
 <xs:restriction base="xs:string">
  <xs:pattern value="[0-9]{6}"/>
 </xs:restriction>
</xs:simpleType>
<xs:complexType name="shiptotype">
 <xs:sequence>
  <xs:element name="name" type="stringtype"/>
  <xs:element name="address" type="stringtype"/>
  <xs:element name="city" type="stringtype"/>
  <xs:element name="country" type="stringtype"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="itemtype">
```

```
 <xs:sequence>
  <xs:element name="title" type="stringtype"/>
  <xs:element name="note" type="stringtype" minOccurs="0"/>
  <xs:element name="quantity" type="inttype"/>
  <xs:element name="price" type="dectype"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="shipordertype">
 <xs:sequence>
  <xs:element name="orderperson" type="stringtype"/>
  <xs:element name="shipto" type="shiptotype"/>
  <xs:element name="item" maxOccurs="unbounded" type="itemtype"/>
 </xs:sequence>
 <xs:attribute name="orderid" type="orderidtype" use="required"/>
</xs:complexType>
<xs:element name="shiporder" type="shipordertype"/>
</xs:schema>
```

The restriction element indicates that the datatype is derived from a W3C XML Schema namespace datatype. So, the following fragment means that the value of the element or attribute must be a string value:<xs:restriction base="xs:string">

The restriction element is more often used to apply restrictions to elements. Look at the following lines from the schema above:

```
<xs:simpleType name="orderidtype">
 <xs:restriction base="xs:string">
  <xs:pattern value="[0-9]{6}"/>
 </xs:restriction>
</xs:simpleType>
```

This indicates that the value of the element or attribute must be a string, it must be exactly six characters in a row, and those characters must be a number from 0 to 9.

XSD String Data Types

String data types are used for values that contains character strings.

String Data Type

The string data type can contain characters, line feeds, carriage returns, and tab characters.

The following is an example of a string declaration in a schema:<xs:element name="customer" type="xs:string"/>

An element in your document might look like this:
<customer>John Smith</customer>
Or
it might look like this:
<customer>   John Smith   </customer>
Note: The XML processor will not modify the value if you use the string data type.

NormalizedString Data Type

The normalizedString data type is derived from the String data type.

The normalizedString data type also contains characters, but the XML processor will remove line feeds, carriage returns, and tab characters.

The following is an example of a normalizedString declaration in a schema:<xs:element name="customer" type="xs:normalizedString"/>

An element in your document might look like this:<customer>John Smith</customer>
Or
It might look like this:
<customer>   John Smith   </customer>

Note: In the example above the XML processor will replace the tabs with spaces.

Token Data Type

The token data type is also derived from the String data type.
The token data type also contains characters, but the XML processor will remove line

feeds, carriage returns, tabs, leading and trailing spaces, and multiple spaces.

The following is an example of a token declaration in a schema:
<xs:element name="customer" type="xs:token"/>

An element in your document might look like this:<customer>John Smith</customer>
Or
It might look like this:
<customer>   John Smith   </customer>
Note: In the example above the XML processor will remove the tabs.

String Data Types

Note that all of the data types below derive from the String data type (except for string itself)!

ENTITIES

ENTITY

ID - A string that represents the ID attribute in XML (only used with schema attributes)

IDREF - A string that represents the IDREF attribute in XML (only used with schema attributes)

IDREFS

Language - A string that contains a valid language id

Name - A string that contains a valid XML name

NCName

NMTOKEN - A string that represents the NMTOKEN attribute in XML (only used with schema attributes)

NMTOKENS

normalizedString - A string that does not contain line feeds, carriage returns, or tabs

QName

String - A string

Token - A string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces

Restrictions on String Data Types

Restrictions that can be used with String data types:

- enumeration

- length
- maxLength
- minLength
- pattern (NMTOKENS, IDREFS, and ENTITIES cannot use this constraint)
- whitespace

XSD Date and Time Data Types

Date and time data types are used for values that contain date and time.

Date Data Type

The date data type is used to specify a date.

The date is specified in the following form "YYYY-MM-DD" where:

YYYY indicates the year

MM indicates the month

DD indicates the day

Note: All components are required!

The following is an example of a date declaration in a schema:<xs:element name="start" type="xs:date"/>

An element in your document might look like this:<start>2002-09-24</start>

Time Zones

To specify a time zone, you can either enter a date in UTC time by adding a "Z" behind the date - like this:

<start>2002-09-24Z</start>

Or

You can specify an offset from the UTC time by adding a positive or negative time behind the date - like this:<start>2002-09-24-06:00</start>

or

<start>2002-09-24+06:00</start>

Time Data Type

The time data type is used to specify a time. The time is specified in the following form "hh:mm:ss" where:

hh indicates the hour

mm indicates the minute

ss indicates the second

Note: All components are required!

The following is an example of a time declaration in a schema:

<xs:element name="start" type="xs:time"/>

An element in your document might look like this:<start>09:00:00</start>

Or

It might look like this:<start>09:30:10.5</start>

Time Zones

To specify a time zone, you can either enter a time in UTC time by adding a "Z" behind the time - like this:<start>09:30:10Z</start>

Or

You can specify an offset from the UTC time by adding a positive or negative time behind the time - like this:

<start>09:30:10-06:00</start>

Or

<start>09:30:10+06:00</start>

DateTime Data Type

The dateTime data type is used to specify a date and a time. The dateTime is specified in the following form "YYYY-MM-DDThh:mm:ss" where:

YYYY indicates the year

MM indicates the month

DD indicates the day

T indicates the start of the required time section

hh indicates the hour

mm indicates the minute

ss indicates the second

Note: All components are required!

The following is an example of a dateTime declaration in a schema:<xs:element name="startdate" type="xs:dateTime"/>

An element in your document might look like this:<startdate>2002-05-30T09:00:00</startdate>

Or

It might look like this:<startdate>2002-05-30T09:30:10.5</startdate>

Time Zones

To specify a time zone, you can either enter a dateTime in UTC time by adding a "Z" behind the time - like this:<startdate>2002-05-30T09:30:10Z</startdate>

Or

You can specify an offset from the UTC time by adding a positive or negative time behind the time - like this:<startdate>2002-05-30T09:30:10-06:00</startdate>

Or

<startdate>2002-05-30T09:30:10+06:00</startdate>

Duration Data Type

The duration data type is used to specify a time interval.

The time interval is specified in the following form "PnYnMnDTnHnMnS" where:

P indicates the period (required)

nY indicates the number of years

nM indicates the number of months

nD indicates the number of days

T indicates the start of a time section (required if you are going to specify hours, minutes, or seconds)

nH indicates the number of hours

nM indicates the number of minutes

nS indicates the number of seconds

The following is an example of a duration declaration in a schema:<xs:element name="period" type="xs:duration"/>

An element in your document might look like this:<period>P5Y</period>

The example above indicates a period of five years.

Or

it might look like this:<period>P5Y2M10D</period>

The example above indicates a period of five years, two months, and 10 days.

Or

It might look like this :<period>P5Y2M10DT15H</period>

The example above indicates a period of five years, two months, 10 days, and 15 hours.

Or

It might look like this :<period>PT15H</period>

The example above indicates a period of 15 hours.

Negative Duration

To specify a negative duration, enter a minus sign before the P:<period>-P10D</period>

The example above indicates a period of minus 10 days.

Date and Time Data Types

Date - Defines a date value

dateTime - Defines a date and time value

duration - Defines a time interval

gDay - Defines a part of a date - the day (DD)

gMonth - Defines a part of a date - the month (MM)

gMonthDay - Defines a part of a date - the month and day (MM-DD)

gYear - Defines a part of a date - the year (YYYY)

gYearMonth - Defines a part of a date - the year and month (YYYY-MM)

time - Defines a time value

Restrictions on Date Data Types

Restrictions that can be used with Date data types:

- ***enumeration***
- ***maxExclusive***
- ***maxInclusive***
- ***minExclusive***
- ***minInclusive***
- ***pattern***
- ***whitespace***

XSD Numeric Data Types

Decimal data types are used for numeric values.

Decimal Data Type

The decimal data type is used to specify a numeric value. The following is an example of a decimal declaration in a schema:

<xs:element name="prize" type="xs:decimal"/>

An element in your document might look like this:<prize>999.50</prize>

Or

it might look like this:<prize>+999.5450</prize>

Or

It might look like this :<prize>-999.5230</prize>

Or

It might look like this:<prize>0</prize>

Or

It might look like this:<prize>14</prize>

Note: The maximum number of decimal digits you can specify is 18.

Integer Data Type

The integer data type is used to specify a numeric value without a fractional component.

The following is an example of an integer declaration in a schema:<xs:element name="prize" type="xs:integer"/>

An element in your document might look like this:<prize>999</prize>

Or

It might look like this :<prize>+999</prize>

Or

It might look like this:<prize>-999</prize>

Or

It might look like this:<prize>0</prize>

Numeric Data Types

Note that all of the data types below derive from the Decimal data type (except for decimal itself)!

Byte - A signed 8-bit integer

Decimal - A decimal value

Int - A signed 32-bit integer

Integer - An integer value

Long - A signed 64-bit integer

negativeInteger - An integer containing only negative values (.., -2, -1.)

nonNegativeInteger - An integer containing only non-negative values (0, 1, 2, ..)

nonPositiveInteger - An integer containing only non-positive values (.., -2, -1, 0)

positiveInteger - An integer containing only positive values (1, 2, ..)

short - A signed 16-bit integer

unsignedLong - An unsigned 64-bit integer

unsignedInt - An unsigned 32-bit integer

unsignedShort - An unsigned 16-bit integer

unsignedByte - An unsigned 8-bit integer

Restrictions on Numeric Data Types

Restrictions that can be used with Numeric data types:

- ***enumeration***
- ***fractionDigits***
- ***maxExclusive***
- ***maxInclusive***
- ***minExclusive***
- ***minInclusive***
- ***pattern***
- ***totalDigits***
- ***whitespace***

XSD Miscellaneous Data Types

Other miscellaneous data types are boolean, base64Binary, hexBinary, float, double, anyURI, QName, and NOTATION.

Boolean Data Type

The boolean data type is used to specify a true or false value.

The following is an example of a boolean declaration in a schema:<xs:attribute name="disabled" type="xs:boolean"/>

An element in your document might look like this:<prize disabled="true">999</prize>

Note: Legal values for boolean are true, false, 1 (which indicates true), and 0 (which indicates false).

Binary Data Types

Binary data types are used to express binary-formatted data.

We have two binary data types:

- base64Binary (Base64-encoded binary data)
- hexBinary (hexadecimal-encoded binary data)

The following is an example of a hexBinary declaration in a schema:<xs:element name="blobsrc" type="xs:hexBinary"/>

AnyURI Data Type

The anyURI data type is used to specify a URI.

The following is an example of an anyURI declaration in a schema:<xs:attribute name="src" type="xs:anyURI"/>. An element in your document might look like this:<pic src="http://www.w3schools.com/images/smiley.gif" />

Note: If a URI has spaces, replace them with %20.

Miscellaneous Data

- **anyURI**
- **base64Binary**
- **boolean**

- **double**
- **float**
- **hexBinary**
- **NOTATION**
- **QName**

Restrictions on Miscellaneous Data Types

Restrictions that can be used with the other data types:

- ***enumeration (a Boolean data type cannot use this constraint)***
- ***length (a Boolean data type cannot use this constraint)***
- ***maxLength (a Boolean data type cannot use this constraint)***
- ***minLength (a Boolean data type cannot use this constraint)***
- ***pattern***
- ***whitespace***

You Have Learned XML Schema, Now What?

XML Schema Summary

You have learned how to use an XML Schema is to define the legal elements of an XML document, just like a DTD. We think that very soon XML Schemas will be used in most Web applications as a replacement for DTDs. You have also learned that the XML Schema language is very rich. Unlike DTDs, it supports data types and namespaces.

XSL Languages

XSL - stands for eXtensible Stylesheet Language. The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Style sheet Language.

CSS = HTML Style Sheets

HTML uses predefined tags and the meanings of the tags are well understood. The <table> element in HTML defines a table - and a browser knows how to display it. Adding styles to HTML elements is simple. Telling a browser to display an element in a special font or color is easy with CSS.

XSL = XML Style Sheets

XML does not use predefined tags (we can use any tag-names we like), and the meaning of these tags are not well understood. A <table> element could mean an HTML table, a piece of furniture, or something else - and a browser does not know how to display it.

XSL describes how the XML document should be displayed!

XSL - More Than a Style Sheet Language

XSL consists of three parts:

1. XSLT - a language for transforming XML documents
2. XPath - a language for navigating in XML documents
3. XSL-FO - a language for formatting XML documents

XSLT - the language for transforming XML documents.

XSLT is a language for transforming XML documents into XHTML documents or to other XML documents.

XPath is a language for navigating in XML documents.

What You Should Already Know

- HTML / XHTML
- XML / XML Namespaces
- XPath

What is XSLT?

- XSLT stands for XSL Transformations
- XSLT is the most important part of XSL
- XSLT transforms an XML document into another XML document
- XSLT uses XPath to navigate in XML documents
- XSLT is a W3C Recommendation
- XSLT = XSL Transformations

XSLT is the most important part of XSL.

XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X) HTML element.

With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

A common way to describe the transformation process is to say that XSLT transforms an XML source-tree into an XML result-tree.

XSLT Uses XPath

XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.

How does it Work?

In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

- XSLT is a W3C Recommendation
- XSLT became a W3C Recommendation 16. November 1999.

XSLT Browsers

- Nearly all major browsers have support for XML and XSLT.

- Netscape
- Opera
- Internet Explorer

XSLT – Transformation

Example study: How to transform XML into XHTML using XSLT.

Correct Style Sheet Declaration

The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>.

Note: <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used!

The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

or

```
<xsl:transform version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

To get access to the XSLT elements, attributes and features we must declare the XSLT namespace at the top of the document.

The xmlns:xsl="http://www.w3.org/1999/XSL/Transform" points to the official W3C XSLT namespace. If you use this namespace, you must also include the attribute version="1.0".

Start with a Raw XML Document

We want to transform the following XML document ("cdcatalog.xml") into XHTML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
```

```
   <company>Columbia</company>
   <price>10.90</price>
   <year>1985</year>
  </cd>
.
.
.
</catalog>
```

Viewing XML Files in Firefox and Internet Explorer: Open the XML file (typically by clicking on a link) - The XML document will be displayed with color-coded root and child elements. A plus (+) or minus sign (-) to the left of the elements can be clicked to expand or collapse the element structure. To view the raw XML source (without the + and - signs), select "View Page Source" or "View Source" from the browser menu.

Viewing XML Files in Netscape: Open the XML file, then right-click in XML file and select "View Page Source". The XML document will then be displayed with color-coded root and child elements.

Viewing XML Files in Opera: Open the XML file, then right-click in XML file and select "Frame" / "View Source". The XML document will be displayed as plain text.

- ***View "cdcatalog.xml"***
- **Create an XSL Style Sheet**

Then you create an XSL Style Sheet ("cdcatalog.xsl") with a transformation template:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
   <h2>My CD Collection</h2>
   <table border="1">
   <tr bgcolor="#9acd32">
    <th align="left">Title</th>
    <th align="left">Artist</th>
   </tr>
   <xsl:for-each select="catalog/cd">
   <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
   </tr>
   </xsl:for-each>
   </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

- ***View "cdcatalog.xsl"***

- ***Link the XSL Style Sheet to the XML Document***

Add the XSL style sheet reference to your XML document ("cdcatalog.xml"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
  <cd>
   <title>Empire Burlesque</title>
   <artist>Bob Dylan</artist>
   <country>USA</country>
   <company>Columbia</company>
   <price>10.90</price>
   <year>1985</year>
  </cd>
.
.
.
</catalog>
```

If you have an XSLT compliant browser it will nicely transform your XML into XHTML.

- ***View the result***

XSLT <xsl:template> Element

An XSL style sheet consists of one or more set of rules that are called templates.
Each template contains rules to apply when a specified node is matched.

The <xsl:template> Element

The <xsl:template> element is used to build templates.
The match attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).
Ok, let's look at a simplified version of the XSL file,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
 <body>
   <h2>My CD Collection</h2>
   <table border="1">
     <tr bgcolor="#9acd32">
       <th>Title</th>
       <th>Artist</th>
     </tr>
     <tr>
       <td>.</td>
       <td>.</td>
     </tr>
   </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

Since an XSL style sheet is an XML document itself, it always begins with the XML declaration:

<?xml version="1.0" encoding="ISO-8859-1"?>.

The next element, <xsl:stylesheet>, defines that this document is an XSLT style sheet document (along with the version number and XSLT namespace attributes).

The <xsl:template> element defines a template. The match="/" attribute associates the template with the root of the XML source document.

The content inside the <xsl:template> element defines some HTML to write to the output.

The last two lines define the end of the template and the end of the style sheet.

The result of the transformation above will look like this:

My CD Collection	
Title	**Artist**
.	.

The result from this example was a little disappointing, because no data was copied from the XML document to the output.

XSLT <xsl:value-of> Element

The <xsl:value-of> element is used to extract the value of a selected node.

The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
 <body>
  <h2>My CD Collection</h2>
  <table border="1">
   <tr bgcolor="#9acd32">
     <th>Title</th>
     <th>Artist</th>
   </tr>
   <tr>
```

```
      <td><xsl:value-of select="catalog/cd/title"/></td>
      <td><xsl:value-of select="catalog/cd/artist"/></td>
    </tr>
  </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

Note: The value of the select attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

The result of the transformation above will look like this:

```
My CD Collection
Title               Artist
Empire Burlesque          Bob Dylan
```

XSLT <xsl:for-each> Element

The <xsl:for-each> element allows you to do looping in XSLT.

The XSL <xsl:for-each> element can be used to select every XML element of a specified node-set:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
    </xsl:for-each>
   </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The value of the select attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

The result of the transformation above will look like this:

```
My CD Collection
Title               Artist
Empire Burlesque          Bob Dylan
Hide your heart           Bonnie Tyler
Greatest Hits       Dolly Parton
Still got the blues       Gary More
Eros                Eros Ramazzotti
One night only            Bee Gees
```

Filtering the Output

We can also filter the output from the XML file by adding a criterion to the select attribute in the <xsl:for-each> element.

```
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
```

Legal filter operators are:

= (equal)

!= (not equal)

&lt; less than

&gt; greater than

Take a look at the adjusted XSL style sheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
   <tr bgcolor="#9acd32">
     <th>Title</th>
     <th>Artist</th>
   </tr>
   <xsl:for-each select="catalog/cd[artist='Bob Dylan']">
   <tr>
     <td><xsl:value-of select="title"/></td>
     <td><xsl:value-of select="artist"/></td>
   </tr>
   </xsl:for-each>
  </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan

XSLT <xsl:sort> Element

The <xsl:sort> element is used to sort the output.

Where to put the Sort Information?

To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
 <body>
  <h2>My CD Collection</h2>
  <table border="1">
   <tr bgcolor="#9acd32">
    <th>Title</th>
    <th>Artist</th>
   </tr>
   <xsl:for-each select="catalog/cd">
   <xsl:sort select="artist"/>
   <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
   </tr>
   </xsl:for-each>
  </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

The select attribute indicates what XML element to sort on.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Romanza	Andrea Bocelli
One night only	Bee Gees
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler

The very best of	Cat Stevens
Greatest Hits	Dolly Parton
Sylvias Mother	Dr.Hook
…	…

XSLT <xsl:if> Element

To put a conditional if test against the content of the XML file, add an <xsl:if> element to the XSL document.

Syntax

```
<xsl:if test="expression">
    ... Some output if the expression is true...
</xsl:if>
```

Where to Put the <xsl:if> Element

To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
      <xsl:if test="price &gt; 10">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
      </xsl:if>
    </xsl:for-each>
   </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The value of the required test attribute contains the expression to be evaluated.

The code above will only output the title and artist elements of the CDs that has a price that is higher than 10.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan
Still got the blues	Gary Moore
One night only	Bee Gees
Romanza	Andrea Bocelli
Black Angel	Savage Rose
1999 Grammy Nominees	Many

XSLT <xsl:choose> Element

The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests.

Syntax<xsl:choose>

```
  <xsl:when test="expression">
    ... some output ...
  </xsl:when>
  <xsl:otherwise>
    ... some output ....
  </xsl:otherwise>
</xsl:choose>
```

Where to put the Choose Condition?

To insert a multiple conditional test against the XML file, add the <xsl:choose>, <xsl:when>, and <xsl:otherwise> elements to the XSL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
      <tr>
        <td><xsl:value-of select="title"/></td>
              <xsl:choose>
          <xsl:when test="price &gt; 10">
            <td bgcolor="#ff00ff">
            <xsl:value-of select="artist"/></td>
          </xsl:when>
          <xsl:otherwise>
            <td><xsl:value-of select="artist"/></td>
          </xsl:otherwise>
        </xsl:choose>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```
The code above will add a pink background-color to the "Artist" column WHEN the price of the CD is higher than 10.

The result of the transformation will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
Greatest Hits	Dolly Parton
Still got the blues	Gary Moore
Eros	Eros Ramazzotti
One night only	Bee Gees
Sylvias Mother	Dr.Hook
Maggie May	Rod Stewart
Romanza	Andrea Bocelli

XSLT <xsl:apply-templates> Element

The <xsl:apply-templates> element applies a template to the current element or to the current element's child nodes. If we add a select attribute to the <xsl:apply-templates> element it will process only the child element that matches the value of the attribute. We can use the select attribute to specify the order in which the child nodes are processed.

Look at the following XSL style sheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
<xsl:template match="cd">
<p>
<xsl:apply-templates select="title"/>
```

```
<xsl:apply-templates select="artist"/>
</p>
</xsl:template>
<xsl:template match="title">
Title: <span style="color:#ff0000">
<xsl:value-of select="."/></span>
<br />
</xsl:template>
<xsl:template match="artist">
Artist: <span style="color:#00ff00">
<xsl:value-of select="."/></span>
<br />
</xsl:template>
</xsl:stylesheet>
```

The result of the transformation will look like this:

My CD Collection

Title: Empire Burlesque

Artist: Bob Dylan

Title: Hide your heart

Artist: Bonnie Tyler

Title: Greatest Hits

Artist: Dolly Parton

(w3schools.com)