[Unit VII: XML]

Web Technology

SA

Introduction::

As we have studied in unit one that HTML is designed to display data. In contrast, XML is designed to transport and store data. XML stands for EXtensible Markup Language and is much like HTML. XML was designed to carry data, not to display data. XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive. **Extensible Markup Language** (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

XML is not a replacement for HTML. XML and HTML were designed with different goals:

- XML was designed to transport and store data, with focus on what data is
- HTML was designed to display data, with focus on how data looks

HTML is about displaying information, while XML is about carrying information.

Maybe it is a little hard to understand, but XML does not DO anything. XML was created to structure, store, and transport information. The following example is a note to Tulsi, from Giri, stored as XML:

```
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk web tech class at Patan!</body>
</note>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body. But still, this XML document does not DO anything. It is just information wrapped in tags. Someone must write a piece of software to send, receive or display it.

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document. That is because the XML language has no predefined tags. However, the tags used in HTML are predefined. HTML documents can only use tags defined in the HTML standard (like <p>, <h1>, etc.). In contrast, XML allows the author to define his/her own tags and his/her own document structure. The XML processor can not tell us which elements and attributes are valid. As a result we need to define the XML markup we are using. To do this, we need to define the markup language's grammar. There are numerous "tools" that can be used to build an XML language – some relatively simple, some much more complex. They include DTD (Document Type Definition), RELAX, TREX, RELAX NG, XML Schema, Schmatron, etc.

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.

XML Usages

XML is used in many aspects of web development, often to simplify data storage and sharing.

XML Separates Data from HTML: If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes. With XML, data can be stored in separate XML files. This way you can concentrate on using HTML for layout and display, and be sure that changes in the underlying data will not require any changes to the HTML. With a few lines of JavaScript code, you can read an external XML file and update the data content of your web page.

XML Simplifies Data Sharing: In the real world, computer systems and databases contain data in incompatible formats. XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data. This makes it much easier to create data that can be shared by different applications.

XML Simplifies Data Transport: One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet. Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

XML Simplifies Platform Changes: Upgrading to new systems (hardware or software platforms), is always time consuming. Large amounts of data must be converted and incompatible data is often lost. XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Makes Your Data More Available: Different applications can access your data, not only in HTML pages, but also from XML data sources. With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc), and make it more available for blind people, or people with other disabilities.

XML Used to Create New Internet Languages: A lot of new Internet languages are created with XML. Here are some examples:

- XHTML
- WSDL (Web Services Description Language) for describing available web services
- WAP and WML (Wireless Markup Language) as markup languages for handheld devices
- RSS (Really Simple Syndication / Rich Site Summary) languages for news feeds
- RDF (Resource Description Framework), a family of w3c spec, and OWL (Web Ontology Language) for describing resources and ontology
- SMIL (Synchronized Multimedia Integration Language) for describing multimedia for the web

XML Tree

XML documents form a tree structure that starts at "the root" and branches to "the leaves". XML documents use a self-describing and simple syntax:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
        <to>Tulsi</to>
        <from>Giri</from>
         <heading>Reminder</heading>
        <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0) and the encoding used (ISO-8859-1 = Latin-1/West European character set). The next line describes the **root element** of the document (like saying: "this document is a note"):

```
<note>
```

The next 4 lines describe 4 **child elements** of the root (to, from, heading, and body):

```
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk the web tech class at Patan!</body>
```

And finally the last line defines the end of the root element:

```
</note>
```

You can assume, from this example, that the XML document contains a note to Tulsi from Giri.

Thus, XML documents must contain a **root element**. This element is "the parent" of all other elements. The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree. All elements can have sub elements (child elements):

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

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters). All elements can have text content and attributes (just like in HTML).

XML Syntax Rules

The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.

1. **All XML Elements Must Have a Closing Tag**. In HTML, some elements may not have to have a closing tag, like;

   ```
   <p>This is a paragraph.
   <br>
   ```

   In XML, it is illegal to omit the closing tag. All elements must have a closing tag:

   ```
   <p>This is a paragraph.</p>
   <br />
   <hello> This is hello </hello>
   ```

2. **XML tags are case sensitive**. The tag <Letter> is different from the tag <letter>. Opening and closing tags must be written with the same case:

   ```
   <Message>This is incorrect</message>
   <message>This is correct</message>
   ```

3. **XML Elements Must be Properly Nested**. In HTML, you might see improperly nested elements:

   ```
   <b><i>This text is bold and italic</b></i>
   ```

   In XML, all elements must be properly nested within each other:

   ```
   <b><i>This text is bold and italic</i></b>
   ```

4. **XML Documents Must Have a Root Element**. XML documents must contain one element that is the **parent** of all other elements. This element is called the **root** element.

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

5. **XML Attribute Values Must be Quoted.** XML elements can have attributes in name/value pairs just like in HTML. In XML, the attribute values must always be quoted. Study the two XML documents below. The first one is incorrect, the second is correct:

```
<note date=06/01/2012>
  <to>Tulsi</to>
  <from>Giri</from>
</note>

<note date="06/01/2012">
  <to>Tulsi</to>
  <from>Giri</from>
</note>
```

   The error in the first document is that the date attribute in the note element is not quoted.

6. **Entity Reference.** Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element. This will generate an XML error:

```
<message>if salary < 1000 then</message>
```

   To avoid this error, replace the "<" character with an **entity reference**:

```
<message>if salary &lt; 1000 then</message>
```

   There are 5 predefined entity references in XML:

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	quotation mark

7. **Comments in XML.** The syntax for writing comments in XML is similar to that of HTML.

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

8. **White-space is preserved in XML**. HTML truncates multiple white-space characters to one single white-space:

   ```
   HTML: Hello          Tulsi
   Output: Hello Tulsi
   ```

   With XML, the white-space in a document is not truncated.

XML Elements

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

An element can contain:

- other elements
- text
- attributes
- or a mix of all of the above...

Consider an example;

```
<bookstore>

  <book category="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>

  <book category="WEB">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>

</bookstore>
```

In the example above, <bookstore> and <book> have **element contents**, because they contain other elements. <book> also has an **attribute** (category="CHILDREN"). <title>, <author>, <year>, and <price> have **text content** because they contain text.

XML Naming Rules

XML elements must follow these naming rules:

- Names can contain letters, numbers, and other characters
- Names cannot start with a number or punctuation character
- Names cannot start with the letters xml (or XML, or Xml, etc)
- Names cannot contain spaces
- Any name can be used, no words are reserved.

Best Naming Practices

- Make names descriptive. Names with an underscore separator are nice: <first_name>, <last_name>.

- Names should be short and simple, like this: <book_title> not like this: <the_title_of_the_book>.

- Avoid "-" characters. If you name something "first-name," some software may think you want to subtract name from first.

- Avoid "." characters. If you name something "first.name," some software may think that "name" is a property of the object "first."

- Avoid ":" characters. Colons are reserved to be used for something called namespaces (more later).

- XML documents often have a corresponding database. A good practice is to use the naming rules of your database for the elements in the XML documents.

- Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software vendor doesn't support them.

XML Elements are Extensible

XML elements can be extended to carry more information. Look at the following XML example:

```
<note>
<to>Tulsi</to>
<from>Giri</from>
```

```
<body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

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

MESSAGE

To: Tulsi
From: Giri

Don't forget to bunk the web tech class at Patan!

Suppose the XML document has been modified by adding some extra information to it like:

```
<note>
<date>2012-01-06</date>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk thee web tech class at Patan!</body>
</note>
```

Should the application break or crash?

No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output. Thus, one of the beauties of XML, is that it can be extended without breaking applications.

XML Attributes

XML elements can have attributes, just like HTML. Attributes provide additional information about an element. In HTML, attributes provide additional information about elements:

```
<img src="computer.gif">
<a href="demo.asp">
```

Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but can be important to the software that wants to manipulate the element:

```
<file type="gif">computer.gif</file>
```

Attribute values must always be quoted. Either single or double quotes can be used. For a person's sex, the person element can be written like this:

```
        <person sex="male">
```

or like this:

```
    <person sex='male'>
```

If the attribute value itself contains double quotes you can use single quotes, like in this example:

```
    <gangster name='Chota "Shotgun" Chetan'>
```

or you can use character entities:

```
    <gangster name="Chota &quot;Shotgun&quot; Chetan">
```

XML Elements vs. Attributes

Take a look at these examples:

```
    <person sex="male">
        <firstname>Sanjay</firstname>
        <lastname>Bhatta</lastname>
    </person>

    <person>
        <sex>male</sex>
        <firstname>Sanjay</firstname>
        <lastname>Bhatta</lastname>
    </person>
```

In the first example sex is an attribute. In the last, sex is an element. Both examples provide the same information. There are no rules about when to use attributes or when to use elements. Attributes are handy in HTML. In XML my advice is to avoid them. Use elements instead.

Writing in different ways

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
    <note date="10/01/2008">
      <to>Tulsi</to>
      <from>Giri</from>
      <heading>Reminder</heading>
```

```
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

A date element is used in the second example:

```
<note>
  <date>10/01/2008</date>
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

An expanded date element is used in the third:

```
<note>
  <date>
    <day>10</day>
    <month>01</month>
    <year>2008</year>
  </date>
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

Restrictions with XML Attributes

Some of the problems with using attributes are:

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

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

XML Attributes for Metadata

Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. This example demonstrates this:

```
<messages>
  <note id="501">
    <to>Tulsi</to>
    <from>Giri</from>
    <heading>Reminder</heading>
    <body>Don't forget to bunk the web tech class at Patan!</body>
  </note>

  <note id="502">
    <to>Giri</to>
    <from>Tulsi</from>
    <heading>Re: Reminder</heading>
    <body>Ok Giri dai !!</body>
  </note>
</messages>
```

The id attributes above are for identifying the different notes. It is not a part of the note itself. In other words, metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.

XML Validation

XML with correct syntax is "Well Formed" XML. XML validated against a DTD is "Valid" XML.

Well Formed XML Documents

A "Well Formed" XML document has correct XML syntax. The syntax rules as described in previous sections are:

- XML documents must have a root element
- XML elements must have a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML attribute values must be quoted

Consider the earlier example;

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

```
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don’t forget to bunk the web tech class at Patan!</body>
</note>
```

Now, a "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don’t forget to bunk the web tech class at Patan!</body>
</note>
```

The DOCTYPE declaration in the example above, is a reference to an external DTD file. The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements. For above example the DTD seems like;

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

W3C supports an XML-based alternative to DTD, called XML Schema:

```
<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>
```

XML schema:

An **XML schema** is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, above and beyond the basic syntactical constraints imposed by XML itself. These constraints are generally expressed using some combination of grammatical rules governing the order of elements, Boolean predicates that the content must satisfy, data types governing the content of elements and attributes, and more specialized rules such as uniqueness and referential integrity constraints.

Technically, a **schema** is an abstract collection of metadata, consisting of a set of **schema components**: chiefly element and attribute declarations and complex and simple type definitions. These components are usually created by processing a collection of **schema documents**, which contain the source language definitions of these components. In popular usage, however, a schema document is often referred to as a schema.

Schema documents are organized by namespace: all the named schema components belong to a target namespace, and the target namespace is a property of the schema document as a whole. A schema document may *include* other schema documents for the same namespace, and may *import* schema documents for a different namespace.

There are languages developed specifically to express XML schemas. The **Document Type Definition (DTD)** language, which is native to the XML specification, is a schema language that is of relatively limited capability, but that also has other uses in XML aside from the expression of schemas. Two more expressive XML schema languages in widespread use are **XML Schema** (with a capital *S*) and **RELAX NG** (REgular LAnguage for XML Next Generation).

There is some confusion as to when to use the capitalized spelling "Schema" and when to use the lowercase spelling. The lowercase form is a generic term and may refer to any type of schema, including DTD, XML Schema (aka XSD), RELAX NG, or others, and should always be written using lowercase except when appearing at the start of a sentence. The form "Schema" (capitalized) in common use in the XML community always refers to W3C XML Schema.

XML Namespace:

XML Namespaces provide a method to avoid element name conflicts. In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications. Consider following examples;

This XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a table (a piece of furniture):

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

An XML parser will not know how to handle these differences.

Thus, **xmlns** tagged **XML namespaces** are used for providing uniquely named elements and attributes in an XML document. They are defined in a W3C recommendation. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved. The XML namespace is a special type of *reserved XML attribute* that you place in an XML tag. The reserved attribute is actually more like a prefix that you attach to any namespace you create. This *attribute prefix* is "**xmlns:**", which stands for XML NameSpace. The colon is used to separate the prefix from your namespace that you are creating.

A *namespace name* is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organisation defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code

The name conflicts in above mentioned example can be handled by using the concept of namespace as a name prefix, as below ;

This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>
```

```
<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

When using prefixes in XML, a so-called **namespace** for the prefix must be defined. The namespace is defined by the **xmlns attribute** in the start tag of an element. The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

```
<root>

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

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

</root>
```

In the example above, the xmlns attribute in the <table> tag give the h: and f: prefixes a qualified namespace. When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

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

```
<root
xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="http://www.w3schools.com/furniture">

<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
```

```
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

The namespace URI is not used by the parser to look up information. The purpose is to give the namespace a unique name. However, often companies use the namespace as a pointer to a web page containing namespace information.

XML schema languages

- **DTD**
- **XML Schema**

Document Type Definition (DTD)

DTD is an approach for defining the structure of XML Document. It is an XML schema language whose purpose is to define legal building blocks of an XML document. A DTD defines the document structure with a list of legal elements and attributes.

Document Type Definition (**DTD**) is a set of *markup declarations* that define a *document type* for SGML-family markup languages (SGML, XML, HTML). DTDs were a precursor to XML schema and have a similar function, although different capabilities.

DTDs use a terse formal syntax that declares precisely which elements and references may appear where in the document of the particular type, and what the elements' contents and attributes are. DTDs also declare entities which may be used in the *instance* document. XML uses a subset of SGML DTD.

We use DTD because 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.

A Document Type Declaration associates a DTD with an XML document. Document Type Declarations appear in the syntactic fragment *doctypedecl* near the start of an XML document. The declaration establishes that the document is an instance of the type defined by the referenced DTD.

DTDs make two sorts of declaration:

- an optional *external subset*
- an optional *internal subset*

The declarations in the internal subset form part of the Document Type Declaration in the document itself. The declarations in the external subset are located in a separate text file.

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

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

If the DTD is declared in an external file, it should be wrapped in a DOCTYPE definition. Here, DTD is present in separate file and a reference is placed to its location in the document. External DTD’s are easy to apply to multiple documents. In case, a modification is to be made in future, it could be done in just one file and the onerous task of doing it for all the documents is omitted. External DTDs are of two types: **private** and **public**.

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

<!DOCTYPE root-element SYSTEM "DTD location">.

For Example, the listed below is the same XML document as above, but with an external DTD.

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tulsi</to>
  <from>Girii</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)>
```

Public external DTDs are identified by the keyword `PUBLIC` and are intended for broad use. Its syntax is: <!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">.

The DTD_name follows the syntax:

"prefix//owner_of_the_DTD//description_of_the_DTD//ISO 639_language_identifier".

For example,

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"

"http://www.w3.org/TR/REC-html40/loose.dtd">

The following prefixes are allowed in the DTD name:

Prefix:	**Definition:**
ISO	The DTD is an ISO standard. All ISO standards are approved.
+	The DTD is an approved non-ISO standard.
-	The DTD is an unapproved non-ISO standard.

Defining Elements:

Elements are the main building blocks of XML documents. In a DTD, elements are declared with an ELEMENT declaration with the following syntax. <!ELEMENT element-name category>
Or
<!ELEMENT element-name (element-content)>

Empty elements are declared with the category keyword EMPTY. Its syntax is: <!ELEMENT element-name EMPTY>. For example, <!ELEMENT br EMPTY>.

Elements with only parsed character data are declared with #PCDATA inside parentheses. Its syntax is: <!ELEMENT element-name (#PCDATA)>. For example, <!ELEMENT from (#PCDATA)>.

Elements with any content are declared with the category keyword ANY, can contain any combination of parsable data. Its syntax is: <!ELEMENT element-name ANY>. For example, <!ELEMENT note ANY>.

Elements with one or more children are declared with the name of the children elements inside parentheses. Its syntax is <!ELEMENT element-name (child1, child2,…)>. For example, <!ELEMENT note (to,from,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.

We can declare **only one occurrence of an element**. Its syntax is: <!ELEMENT element-name (child-name)>. For example, <!ELEMENT note (message)>. This example declares that the child element "message" must occur once, and only once inside the "note" element.

We can also declare **minimum one occurrence of an element**. Its syntax is <!ELEMENT element-name (child-name+)>. For 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.

Note: We can use * in place of + to declare zero or more occurrence of an element. We can use ? in place of + to declare zero or one occurrence of an element

We can also declare **either/or content.** For example, <!ELEMENT note (to,from,header,(message|body))>. This example declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

We can declare **mixed content**. For example, <!ELEMENT note (#PCDATA|to|from|header|message)*>. This example declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

Defining Attributes

In a DTD, attributes are declared with an **ATTLIST** declaration. An attribute declaration has the following syntax:

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

For example,

<!ATTLIST payment type CDATA "check">

And its XML example is

<payment type="check" />

The **attribute-type** can be one of the following:

Type	**Description**
CDATA	The value is character data (text that doesn’t contain

	markup)
(*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 separated by whitespace
ENTITY	The name of an entity (which must be declared in the DTD)
ENTITIES	The value is a list of entities, separated by whitespace
NOTATION	The value is a name of a notation (which must be declared in the DTD)
xml:	The value is a predefined xml value

The **default-value** can be one of the following:

Value	**Explanation**
Value	The default value of the attribute. For example, <!ATTLIST square width CDATA "0">
#REQUIRED	The attribute is required. For example, <!ATTLIST person number CDATA #REQUIRED>
#IMPLIED	The attribute is not required (optional). For example, <!ATTLIST contact fax CDATA #IMPLIED>
#FIXED *value*	The attribute value is fixed. For example, <!ATTLIST sender company CDATA #FIXED "Microsoft">

A Default attribute value:

Example:
DTD
<!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>

Example:
DTD:
<!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">

Example:

DTD:
<!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>

Example:

DTD:
<!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.

DTD Examples:

```
<!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>

]>
```

XML Schema

XML Schema is a XML schema language which is an alternative to DTD. XML Schema is an XML-based alternative to DTD. Unlike DTD, XML Schemas has support for data types and namespaces. The XML Schema language, also referred to as XML Schema Definition (XSD), is used to define XML schema.

An XML Schema:

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

XML Schemas are the successors of DTDs. In near future, XML Schemas will be used in most Web applications as a replacement for DTDs because of the following 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**

DTDs are better for text-intensive applications, while schemas have several advantages for data-intensive workflows. Schemas are written in XML and thusly follow the same rules, while DTDs are written in a completely different language.

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 code 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: .**

The code 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.

The code fragment **xmlns="http://www.w3schools.com"** indicates that the default namespace is "http://www.w3schools.com".

The code 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:

XML documents can have a reference to an XML Schema. For example consider the following “note.xml” file. This file has a reference the “note.xsd” 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>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The code 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.

The code fragment **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** is the namespace.

In the code fragment **xsi:schemaLocation="http://www.w3schools.com note.xsd"**, there are two attribute values. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace.

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>
```

Here, 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.

XSD Simple Type: Consists of simple elements and attributes.

XSD Simple Elements:

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

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

For Example;

Consider the XML elements;

```
<lastname>Bhatta</lastname>
<age>42</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"/>
```

XSD Attributes:

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

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.

XSD Restrictions/ Facets:

1. **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>
```

2. **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>
```

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

3. 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 "zipcode" 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="zipcode">
  <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>
```

4. 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>
```

5. 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

Constraint	**Description**
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 Types:

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

There are four kinds of complex elements:

- empty elements
- elements that contain only other elements
- elements that contain only text
- elements that contain both other elements and text

Note: 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>Sanjay</firstname>
  <lastname>Bhatta</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="Nepali">03/09/2099</date> ....
</description>
```

How to Define a Complex Element

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

```
<employee>
  <firstname>Sanjayfirstname>
  <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.

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>
```

<u>Types of XSD Complex Elements</u>

1. **XSD Empty Element**

   An empty complex element cannot have contents, only attributes. Consider 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 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>
```

2. **XSD Elements only**
   An "elements-only" complex type contains an element that contains only other elements. Consider an XML element "person", that contains only other elements:

```
<person>
  <firstname>Sanjay</firstname>
  <lastname>Bhatta</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>
```

3. XSD Text only Elements

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

Note: You can 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>
```

4. XSD Mixed Content (that contain other element and text)

A mixed complex type element can contain attributes, elements, and text. Consider an XML element, "ordernote", that contains both text and other elements:

```
<ordernnote>
  Dear Mr.<name>SA</name>.
  Your gift order for the valentine day with order id
<orderid>9999</orderid>
  will be shipped on <shipdate>2012-02-13</shipdate>.
</ordernnote>
```

The following schema declares the "ordernote" element:

```
<xs:element name="ordernote">
  <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 "ordernote", 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 "ordernote" element.

We could also give the complexType element a name, and let the "ordernote" 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="ordernote" type="ordertype"/>

<xs:complexType name="ordertype" 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 Indicators:

XSD indicators are used to control how elements are to be used in documents with indicators. There are seven indicators:

1. ***Order indicators:*** They contain;

   - All
   - Choice
   - Sequence

2. ***Occurrence indicators:*** They include;

   - maxOccurs
   - minOccurs

3. ***Group indicators:*** They contain;

   - Group name
   - attributeGroup name

1. **Order Indicators:** Order indicators are used to define the order of the elements.

All Indicator

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>
```

2. 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.

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

Consider an 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>Anjolina</full_name>
  <child_name>Janet</child_name>
</person>

<person>
  <full_name>Dhritrasta</full_name>
  <child_name>Duryodhan</child_name>
  <child_name>Dushasan</child_name>
  <child_name>Kushashan</child_name>
  <child_name>Sushasan</child_name>
</person>

<person>
  <full_name>Bhismapitamaha</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>
```

3. **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 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>Ram</firstname>
  <lastname>Bhagwan</lastname>
  <children>
  <childname>Luv</childname>
 </children>
</person>

<person>
 <firstname>Harry</firstname>
 <lastname>Porter</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 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>Sita</firstname>
  <lastname>Mata</lastname>
</person>

<person gender="male">
  <firstname>Ram</firstname>
  <lastname>Bhagwan</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.