Chapter-3
The Client Tier

XML

Introduction to XML

- ☞ XML stands for Extensible Markup Language created by the World Wide Web Consortium (W3C) to define a syntax for encoding documents that both humans and machines could read. It does this through the use of tags that define the structure of the document, as well as how the document should be stored and transported.
- ☞ It's probably easiest to compare it to another markup language with which you might be familiar-the Hyper Text Markup Language (HTML) used to encode web pages. HTML uses a predefined set of markup symbols (short codes) that describe the format of content on a web page.
- ☞ Some important point remembers that:
  - ❖ Xml (eXtensible Markup Language) is a markup language.
  - ❖ XML is designed to store and transport data.
  - ❖ Xml was released in late 90's. It was created to provide an easy to use and store self-describing data.
  - ❖ XML became a W3C Recommendation of February 10, 1998.
  - ❖ XML is not a replacement for HTML.
  - ❖ XML is designed to be self-descriptive.
  - ❖ XML is designed to carry data, not to display data.
  - ❖ XML tags are not predefined. You must define your own tags.
  - ❖ XML is platform independent and language independent.

The design goal of XML

- ☞ The design goal of XML are as follows:
  - I. XML shall be straightforwardly usable over the Interne.
  - II. XML support a wide variety of applications.
  - III. XML shall be compatible with SGML.
  - IV. It shall be easy to write programs which process XML documents.
  - V. The number of optional features in XML in to be kept to the absolute minimum, ideally zero.
  - VI. XML documents should be human-legible and reasonably clear.
  - VII. The XML design should be prepared quickly.
  - VIII. The design of XML shall be formal and concise.
  - IX. XML documents shall be easy to create.

The difference between XML and HTML

I. XML is not the replacement for HTML.

II. XML and HTML were design with different goals:
   a. XML was designed to describe data, with a focus on what data is
   b. HTML was design to display data, with a focus on how data looks
III. HTML is about displaying information, while XML is about carrying information.

Features and Advantages of XML

- ☞ XML is widely used in the era of web development. It is also used to simplify data storage and data sharing.

1. **XML separates data from HTML**
2. **XML simplifies data sharing**
3. **XML simplifies data transport**
4. **XML simplifies Platform change**
5. **XML increase data availability**
6. **XML can be used to create new Internet languages**

1. **XML separates data from HTML**

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

2. **XML simplifies data sharing**

- ☞ In the real world, computer systems and database 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 application.

3. **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.

4. **XML simplifies Platform change**

- ☞ Upgrading to new system (hardware or software platforms), is always time consuming.
- ☞ Large amount 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.

5. XML increase data availability

- ☞ 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 machine" (Handheld computers, voice machines, news feeds etc.) and make it more available for blind people, or people with other disabilities.

6. XML can be used to create new Internet languages

- ☞ A lot of new Internet languages are created with XML. Some examples are:
  - ❖ **XHTML**
  - ❖ **WSDL** for describing available web services
  - ❖ **WAP** and **WML** as markup languages for handheld devices
  - ❖ **RSS** languages for news feeds
  - ❖ **RDF** and **OWL** for describing resources and ontology
  - ❖ **SMIL** for describing multimedia for the web

Structure of XML Document

- ☞ XML Elements are as follows
  - ❖ XML documents must contain a root element. This element is "the parent" of all other element.
  - ❖ The element 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)
  - ❖ XML element contents are part of the basic document contents. That are store information data.
  - ❖ XML elements are represented by a tags.

Rules for Building Good XML

Rule-1: All XML Must Have a Root Element

- ☞ A root element is simply a set of tags that contains your XML content.

Eg.

```
<book>
        <author>Roshan Bhushal</author>
        <author>Ramesh Singh Saud</author>
        <author>Basant Chapaigai</author>


</book>
```

Rule-2: All Tags Must Be Closed

- When a tag is declared (opened), it must also be closed. Any unclosed tags will break the code. Even tags that don't need to be closed in HTML must be closed in XML or XHTML. To open a tag, type the name of the element between less-than (<) and greater-than (>) characters, like this opening tag.
  Eg.
  <author>Roshan Bhushal</author>
  Computer Essentials
  <hr/>

Rule-3: All Tags Must Be Properly Nested

- When you insert one tag within another pay attention to the order in which you open each tag, and then close the tags in the reverse order.
- If you open element A and then element B, you must first close B before closing A. Even HTML tags that usually will work without a strict structure must follow the stricter XML rules when they are used within an XML file.
  Eg.
  <A><B>Text Here</B></A>
  Text Here

Rule-4: Tag Names Have Strict Limits

- The tag names can't start with the letters *xml* a number, or punctuation, except for the underscore character (_).
- The letter XMl are used in various commends and can't start your tag name. Number and punctuation also aren't allowed in the beginning of the tag name.
  Eg.
  <author>

  <_author>

Rule-5: Tag Name are Case Sensitive

- Uppercase and lowercase matter in XMl. Opening and closing tags must match exactly.
  For example:
  <ROOT>, <Root> and <root> all are different.

  <author>Hemant Baral</author>
  <AUTHOR>Hemant Baral</AUTHOR>

Rule-6: Tag Names Cannot Contain Spaces

- ☞ Spaces in tag names can cause all sorts of problems with data-intensive applications, so they are prohibited in XML.

Rule-7: Attribute Values Must Appear Within Quotes

- ☞ Attribute values modify a tag or help identify the type of information being tagged.
- ☞ If you are a web designer, you may be used to the flexibility of HTML, in which some attributes don't require quotes. In XML, all attribute values must appear within quotes.
  For example:
  <chapter number= "1">
  <artist title= "author" nationality= "Nepali">

Rule-8: White Space Is Preserved

- ☞ If you are in the habit of adding extra spaces and hard returns in your HTML code, watch out such spacing is honored by XML and can play havoc with your applications. Use extra spacing judiciously.

Rule-9: Avoid HTML Tags (Optional)

- ☞ Because you can name tags anything you want, you want, you could use tags reserved for HTML markup, such as <h1>, <p>, <li> and so on.
- ☞ Although permissible in XML, avoid using such tag names unless you want the data to be formatted that way when it's viewed in a browser window.

XML Element Name must be follow this things

- ❖ Element names must be alphabetic or numeric character contains.
- ❖ Element name can't have white spaces contains and
- ❖ Name can't start with a capital letter, numeric or mixed letter.

- ☞ If element contents absence (empty) then you can write element following two way to represent valid standard.
  <element/>
  <element></element>

Nested Syntax of XML Element

- ☞ The nested syntax of XML element as follows:

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

Note:
The terms parent, child and siblings 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 Attributes

- ☞ XML element can have attributes to identify elements.
- ☞ XML attributes specified by **name="value"** pair inside the starting element. XML attribute values must be **quoted**.
- ☞ XML attributes enhance the properties of the elements.
- ☞ XML standard specifies element may have define multiple attributes along with unique attribute name.

<onte id="1" type="daily">

</note>

Avoiding XML Attributes

- ❖ Attributes cannot contain multiple values but child elements can have multiple values.
- ❖ Attributes cannot contain tree structure but child element can.
- ❖ Attributes are not easily expandable. If you want to change in attribute's values in the future, it may be complicated.
- ❖ Attributes cannot describe structure but child elements can.
- ❖ Attributes are more difficult to be manipulated by program code.
- ❖ Attributes values are not easy to test against a DTD, which is used to define the legal elements of an XML document.

XML Comments

- ☞ XML comments are just like HTML comments. We know that the comments are used to make codes more understandable other developers.
- ☞ XML comments add notes or lines for understanding the purpose of an XML comments are necessary.

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

<!—Students marks are uploaded by months--->
<students>

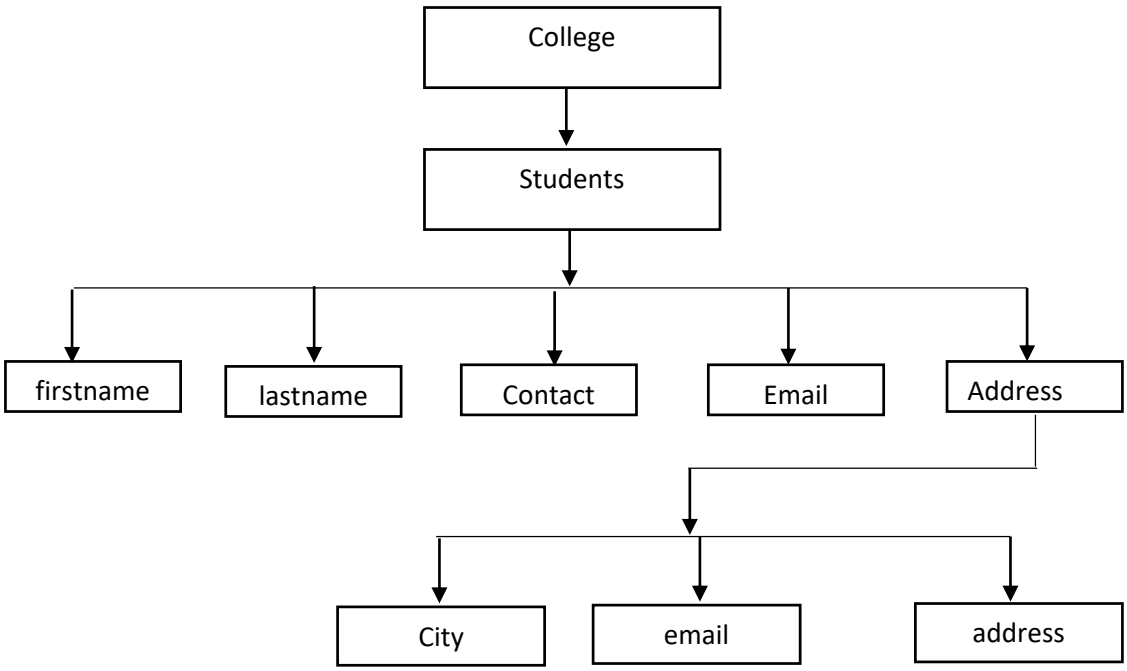
```
    <student>
        <name> Hemant</name>
        <marks>70</marks>
    </student>
    <student>
    <name> Ramesh</name>
    <marks>60</marks>
    </student>
</students>
```

Rules for Adding XML Comments

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

XML Tree Structure

- ☞ An XML document has a self-descriptive structure. It forms a tree structure which is referred as an XML tree. The tree structure makes easy to describe an XML document.
- ☞ A tree structure contains root element (as parent), child element and so on. It is very easy to traverse all succeeding branches and sub-branches and leaf nodes starting from the root. Figure below illustrate the tree structure as:



Example:

```
<?xml version=”1.0” ?>
<college>
    <student>
        <firstname>Ramchandra</firstname>
        <lastname>Paudel</lastname>
        <contact>9841980049</contact>
        <email>ramchandra@gmail.com</email>
        <address>
            <city>Kathmandu</city>
            <state>Bagmati</state>
            <postalcode>44600</postalcode>
        </address>
    </student>
</college>
```

XML Namespaces

- XML Namespaces is primary purpose to distinguish between duplicate elements and attribute names.
- XML data has to be exchanged between several applications. Same tag name may have different meanings in different applications. So It’s create confusion on exchanging documents.
- Specifying prefix name to an element or attribute names to avoid this confusion.
  <prefix_name:element_name>
- Some important point remember that when writing XML namespace as:
  - Elements and attributes name along with namespace have exactly one colon.
  - Colon before text perfi name and colon after text element or attribute name.
  - Each and every prefix name is associated with one URL.

- ❖ Prefix name associated with the same URL are in the same namespace.
- ❖ Fully qualified name including prefix, colon is called the XML qualified name.

☞ Prefixes are bind to a namespace URL using **xmlns:prefix** attribute to the prefixed element.
Eg.
<r:student xmlns:r=http://www.way2tutorial.com/xml/></r:student>

Name Conflicts Example

☞ Following example XML data for storing student marks,

```
<student>
<result>
        <name>Rajan Shrestha</name>
        <sgpa>8.1</sgpa>
        <sgpa>8.4</sgpa>
</result>
<cv>
        <name>Rajan Shrestha</name>
        <cgpa>8.4</cgpa>
</cv>
</student>
```

Explanation:

In the above example, both <result> and <cv> have the same <cgpa> element, so XML parser doesn't know which one is parse. That's why XML namespaces is used for mapping between an element prefix and a URI (Uniform Resource Identifier). XML namespace URI not a point to an information about the namespace but they are identify unique elements.

Convert the Name Conflict to XML Namespace

☞ We are specify prefix name as per different element.

Xmlns attribute with XML namespaces

☞ Specify all XML namespaces within the root element. Specification given for that element is valid for all elements occurring within scope.
☞ The namespace declaration Syntax: xmlns:prefix_name="URI".
Eg.

```
<s:student xmlns:s=http://www.way2tutorial.com/some_urli1"
                Xmlns:res=http://www.way2tutorial.com/some_url2>
<r:result>
        <r:name>Rajan Shrestha</r:name>
        <r:sgpa>8.1</r:sgpa>
        <r:cgpa>8.4</r:cgpa>
</r:result>
<res:cv>
        <res:name>Rajan Shrestha</res:name>
        <res:cgpa>8.4</res:cgpa>
</res:cv>
</s:student>
```

DTD (Document Type Definition)

- DTD stands for Document Type Definition.
- It allows you to create rules for the elements within your XML documents. Although XML itself has rules, the rules defined in a DTD are specific to your own needs.
- So, for an XML document to be well-formed, it needs to use correct XML syntax, and it needs to conform to its DTD or schema.

Do I Need to Create a DTD?

- If you have created your own XML document using predefined elements, and / or entities, then you should create a DTD.
- If you are creating an XML document using predefined elements, attributes, entities (i.e. ones that have been created by someone else), then a DTD should already exist. All you need to do is like to the DTD using the DOCTYPE declaration.

What's In a DTD?

- A DTD consists of a list of syntax definitions for each element in your XML document.
- When you create a DTD, you are creating the syntax rules for any XML document that uses the DTD. You are specifying which element names can be included in the document, the attributes that each element can have, whether or not these are required or optional and more.

DTD <!DOCTYPE>

- ☞ To use a DTD within your XML document, you need to declare it. The DTD can either be internal (written into the same document that it's being used in), or external (located in another document).
- ☞ You declare a DTD at the top of your XML document (in the prolog) using the !DOCTYPE declaration. The basic syntax is:
  <!DOCTYPE rootname [dtd]>
  Where, rootname is the root element and [DTD] is the actual definition.

DTD Variations

<!DOCTYPE rootname [DTD]>

- ☞ This is an internal DTD (the DTD is defined between the square brackets within the XML document).
  <!DOCTYPE tutorials [
  <!ELEMENT tutorials (tutorial)+>
  <!ELEMENT tutorial (name,url)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT url (#PCDATA)>
  <!ATLIST tutorials type CDATA #REQUIRED>
  ]>
  <!DOCTYPE rootname SYSTEM URL>
  <!DOCTYPE rootname SYSTEM URL>
- ☞ The keyword SYSTEM indicates that it's a private DTD (not for public distribution).
- ☞ The presence of URL and [DTD] together indicates that this is both an external and internal DTD (part of the DTD is defined in a document located at the URL, the other part is defined within the XML document).

<!DOCTYPE tutorials SYSTEM "tutorials.dtd">

<!DOCTYPE rootname SYSTEM URL [DTD]>

- ☞ The keyword SYSTEM indicates that it's a private DTD (not for public distribution).
- ☞ The presence of URL and [DTD] together indicates that this is both an external and internal DTD (part o the DTD is defined in a document located at the URL, the other part is defined within the XML document).
  <!DOCTYPE tutorials SYSTEM "tutorials.dtd" [

<ELEMENT tutorial (summary)>
<!ELEMENT summery (#PCDATA)>
]>

<!DOCTYPE rootname PUBLIC identifier URL>

- The keyword **PUBLIC** indicates that it's a public DTD (for public distribution).
- The presence of URL indicates that this is an external DTD (the DTD is defined in a document located at the URL).
- The *identifier* indicates the *formal public identifier* and is required when using a public DTD.

<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Transitional// EN" http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>

<!DOCTYPE rootname PUBLIC identifier URL[DTD]>

- The key word PUBLIC indicates that it's a public DTD (for public distribution).
- The presence of URL and [DTD] together indicates that this is both an external and internal DTD (part of the DTD is defined in a document located at the URL, the other part is defined within the XML document).
- The **identifier** indicates the **formal public identifier** and is required when using a public DTD.

<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Transitional//EN" http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE rootname PUBLIC identifier URL[DTD]>

- The keyword PUBLIC indicates that it's a public DTD (for public distribution).
- The presence of URL and [DTD] together indicates that this is both an internal and internal DTD (part of the DTD is defined in a document located at the URL, the other part is defined within the XML document).
- The ***identifier*** indicates the formal public ***identifier*** and is required when using a public DTD.

<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Transitional//EN" http://www.w3.org/TR/xhtml1/DTD/xhtml1-transational.dtd" [
<!ELEMENT tutorials (tutorial)+>
<!ELEMENT tutorial (name, url)>
<!ELEMENT name (#PCDATA)>

<!ELEMENT url (#PCDATA)>
<!ATTLIST tutorials type CDATA #REQUIRED>
]>

Internal DTD

- Whether you use an external or internal DTD, the actual syntax for the DTD is the same – the same code could just as easily be part of an internal DTD or an external one. The only difference between internal and external is in the way it's declared with DOCTYPE.
- Using an internal DTD, the code is placed between the DOCTYPE tags (eg. **<!DOCTYPE tutorials [and]>**

For example Internal DTD

- This is an example of an internal DTD. It's internal because the DTD is included in the target XML document:

```
<?xml version="1.0" standalone="yes">
<!DOCTYPE tutorials [
<!ELEMENT tutorials (tutorial)+>
<!ELEMENT tutorial (name, url)>
<! ELEMENT name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ATTLIST tutorials type CDATA #REQUIRED>
]>
<tutorials type="web">
        <tutorial>
        <name>XML Tutorial </name>
        <url>https://www.quackit.com/xml/tutorial</url>
        </tutorial>
        <tutorial>
                <name>HTML Tutorial</name>
                <url>https://www.quakit.com/html/tutorial</url>
        <tutorial>

</tutorials>
```

External DTD

- An external DTD is one that resides in a separate document.
- To use the external DTD, you need to link to it from your XML document by providing the URI of the DTD file. This URI is typically in the form of a URL.
- The URL can point to a local file using a relative reference or a remote one (e.g. using HTTP) using an absolute reference.

Example of External DTD

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE tutorials SYSTEM "tutorials.dtd">
<tutorials type="web">
        <tutorial>
                <name>XML Tutorial</name>
                <url>https://www.quackit.com/xml/tutorial</url>
        </tutorial>
        <tutorial>
        <name>HTML Tutorial</name>
        <url>https://www.quckit.com/html/tutorial</url>
        </tutorial>
</tutorials>
```

Using the above XML document as an expel, here's an example, here's an example of what tutorials.dtd (the external DTD file) could look like. Note that the external DTD file doesn't need the DOCTYPE declaration – it is already o the XML file that is using this DTD.

```
<!ELEMENT tutorials (tutorial)+>
<!ELEMENT tutorial (name,url)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ATTLIST tutorials type CDATA #REQUIRED>
```

Combined DTD

- Combined DTD can use both an internal DTD and an external one at the same time.
- This could be useful if you need to adhere to a common DTD, but also need to define your own definitions locally.

Example:

```
<?xml version="1.0 standalone="no"?>
<!DOCTYPE tutorials SYSTEM "tutorials.dtd" [
```

```
<!ELEMENT tutorial (summery)>
<!ELEMENT summary (#PCDATA)>
]>
<tutorials>
        <tutorial>
                <name>XML Tutorial</name>
                <url>http://www.quackit.com/xml/tutorial</url>
                <summary>Best XML tutorial on the web</summary>
        </tutorial>
        <tutorial>
                <name>HTML Tutorial </name>
                <url>https://www.quackit.com/html/tutorial</url>
                <summary>Best HTML tutorial on the web!</summary>
        </tutorial>
</tutorials>
```

Explanation:
In the above example using both an external DTD and an internal one for the same XML document. The external DTD resides in tutorials.dtd and is called first in the DOCTYPE declaration. The internal DTD follows the external one but still resides within the DOCTYPE declaration:
Here I have added a new element called summary. This element must be present under the tutorial element. Because this element hasn't been defined in the external DTD, I need to define it internally. Once again, we are setting the standalone attribute to no because we rely on an external resource.

DTD Formal Public Identifier (FPI)

- ☞ When declaring a DTD available for public use, you need to use the **PUBLIC** keyword within your DOCTYPE declaration.
- ☞ When you use the **PUBLIC** keyword, you also need to use an FPI (which stands for Formal Public Identifier).

  FPI Syntax:
  An FPI is made up of four fields, each separated by double forward slashes (//):
  Field 1//field 2// field 3 // field 4

  FPI example
  Here's a real life example of an FPI. In this case, the DTD was created by the W3C for XHTML: -//W3C//DTD XHTML 1.0 Transitional//EN

FPI Fields

- An FPI must contain the following fields:

Field	Example	Description
Separator	//	This is used to separate the different fields of the FPI.
First Field	-	Indicates whether the DTD is connected to a formal standard or not. If the DTD hasn't been approved (for example, you have defined the DTD yourself), use a hyphen(-). If the DTD has been approved by a non standards body, use a plus sign "+". If the DTD has been approved by a formal standards body this field should be a reference to the standard itself.
Second Field	W3C	Holds the name of the group (or person) responsible for the DTD. The above example is maintained by the W3C, so "W3C" appears in the second field.
Third field	DTD XHTML 1.0 Transitional	Indicates the type of document that is being described. This usually contains some form of unique identifier (such as a version number).
Fourth field	EN	Specifies the language that the DTD uses. This is achieved by using the two letter identifier for the language (i.e. for English, use "EN")

FPI DOCTYPE Syntax

- When using a public DTD, place the FPI between the PUBLIC keyword and the URI/URL.
  <!DOCTYPE rootname PUBLIC FPI URL>

FPI DOCTYPE Example
You can see an example of an FPI in the following DOCTYPE declaration (the FPI is in bold):
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Transitional//EN" http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>

DTD Elements

- Creating a DTD is quite straight forward. It's really just a matter of defining your elements, attributes and entities.
- To define an element in DTD, you use the <ELEMENT> declaration. The actual contents of <!ELEMENT> declaration will depend on the syntax rules you need to apply to your element.

Basic Syntax

The <!ELEMENT> declaration has the following syntax:
<!ELEMENT element_name content_model>
Where **element_name** is the name of the element you are defining. The **content model** could indicate a specific rule, data or another element.

- ❖ If it specifies a rule, it will be set to either ANY or EMPTY.
- ❖ If specifies data or another element, the data type/element name needs to be surrounded by brackets (i.e. (tutorial)or (#PCDATA)).

☞ The following examples show you how to use this syntax for defining your elements.

Plain Text

☞ If an element should contain plain text, you define the element using #PCDATA. Where PCDATA stands for Parsed Character Data and is the way you specify non-markup text in your DTDs.

☞ Using this example -<name>XML Tutorial</name> - the XML Tutorial part is the PCDATA. The other part consists of markup.

Syntax:

<!ELEMENT element_name (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<name>XML Tutorial</name>

Unrestricted Elements

☞ If it doesn't matter what your element contains, you can create an element using the content_model of ANY. Note that doing this removes all syntax checking, so you should avoid using this if possible. You are better off defining a specific content model.

<!ELEMENT element_name ANY>
<!ELEMENT tutorials ANY>

Empty Elements

☞ You might remember that an empty element is one without a closing tag. For example, in XHTML, the </br> and </img> tags are empty elements. Here is how you define an empty element:

Syntax:

<!ELEMENT element_name (child_element_name, child_element_name,……)>
<!ELEMENT tutorial (name, url)>
<tutorials>
    <tutorial>

```
        <name>      </name>
        <url>       </url>
        </tutorial>
</tutorials>
```

DTD ELEMENT OPERATORS

- ☞ An element (tutorial) must contain one instance of another element (tutorial).
- ☞ This is fine if there only needs one instance of tutorial, but what if we didn't want a limit. What if the tutorials element should be able to contain any number of tutorial instances? Fortunately we can do that using DTD operators.
- ☞ Here is a list of operators/ syntax rules we can use when defining child elements as:

Operator	Syntax	Description
+	a^+	One or more occurrences of ***a***
*	a^*	Zero or more occurrences of ***a***
?	*a?*	Either a or nothing
,	*a,b*	***a*** followed by ***b***
|	*a |b*	***a*** or ***b***
0	(exception)	An expression surrounded by parentheses is treated as a unit and could have any one of the following suffixes ?,*, or +.

Zero or More

- ☞ To allow zero or more of the same child element, use an asterisk (*):

  Syntax:

  <!ELEMENT element_name (child_element_name*)>

  Example:

  <!ELEMENT tutorials (tutorial*)>

One or More

- ☞ To allow one or more of the same child element, use a plus sign (+):

  Syntax:

  <!ELEMENT element_name (child_element_name+)>

  Example:

  <!ELEMENT tutorials (tutorial+)>

Zero or One

- ☞ To allow either zero or one of the same child elements, use a question mark (?):

Syntax:
<!ELEMENT element_name (child_element_name?)>
Example:
<!ELEMENT tutorials (tutorial?)>

Choices

- You can define a choice between one or another element by using the pipe (|) operator. For example, if the tutorial element requires a child called either name, title, or subject (but only one of these), you can do the following:
  Syntax:
  <!ELEMENT element_name (choice_1 | choice_2 | choice_3)>
  Example:
  <!ELMENT tutorial (name | title | subject)>

Mixed Content

- You can use the pipe (|) operator to specify that an element can contain both PCDATA and other elements:
  Syntax:
  <!ELEMENT element_name (#PCDATA | child_element_name)>
  Example:
  <!ELEMENT tutorial (#PCDATA | name | title | subject)*>

DTD Operators with Sequences

- You can apply any of the DTD operators to a sequence:
  Syntax:
  <!ELEMENT element_name (child_element_name
  dtd_operator, child_element_name dtd_operator, ……..)>
  Example:
  <!ELEMENT tutorial (name+, url?)>

  In the above example allows the tutorial element to contain one or more instance of the name element and zero or one instance of the url element.

Subsequences

- You can use parentheses to create a subsequence (i.e. a sequence within a sequence). This enables you to apply DTD operators to a subsequence:
  Syntax:
  <!ELEMENT element_name ((sequence) dtd_operator sequence)>

Example:

<!ELEMENT tutorial ((author, rating?)+ name, url*)>

In the above example specifies that the tutorial element can contain one or more author elements, with each occurrence having an optional rating element.

DTD ATTRIBUTES

- Just as you need to define all elements in your DTD, you also need to define any attributes they use.
- You use the **<!ATTLIST>** declaration to define attributes in your DTD.

  Syntax:

  You use a single <!ATTLIST> declaration to declare all attributes for a given element. In other words, for each element (that contains attributes), you only need one <!ATTLIST> declaration.
- The <!ATTLIST> declaration has the following syntax:

  <!ATTLIST element_name
  attribute_name TYPE DEFAULT_VALUE
  attribute_name TYPE DEFAULT_VALUE
  attribute_name TYPE DEFAULT_VALUE

  …….>

Here, element_name fefers to the element that you are defining attributes for attribute_name is the name of the attribute that you are declaring TYPE is the attribute type and DEFAULT_VALUE is it's default value.

Example:

<!ATTLIST tutorial published CDATA "NO">

Here, we are defining an attribute called **published** for the **tutorial** element. The attribute's type is **CDATA** and it's default value is **NO.**

DTD ATTRIBUTE DEFAULT VALUES

- The attribute TYPE field can be set to one of the following values:

Values	Description
Value	A simple text value, enclosed in quotes.
#IMPLIED	Specifies that there is no default value for this attribute and that the attribute is optional.
#REQUIRED	There is no default value for this attribute, but a value must be assigned.

#FIXED *value*	The # FIXED part specifies that the value must be the value provided. The *value* part represents the actual value.

Where,

Value

- ☞ You can provide an actual value to be the default value by placing it in quotes.
  Syntax:
  <!ATTLIST element_name attribute_name CDATA "default_value">
  Example:
  <!ATTLIST tutorial published CDATA "NO">

#REQUIRED

- ☞ The #REQUIRED keyword specifies that you won't be providing a default value, but that you require that anyone using this DTD does provide one.
  Syntax:
  <!ATTLIST element_name attribute_name CDATA #REQUIRED>

  Example:
  <!ATTLIST tutorial published CDATA #REQUIRED>

#IMPLIED

- ☞ The **#IMPLIED** keyword specifies that you won't be providing a default value and that the attribute is optional for users of this DTD.
  Syntax:
  <!ATTLIST element_name attribute_name CDATA #FIXED "value">

  Example:
  <!ATTLIST tutorial language CDATA #FIXED "EN">

#FIXED

- ☞ The #FIXED keyword specifies that you will provide value and that's the only value that can be used by users of the DTD.
  Syntax:
  <!ATTLIST element_name attribute_name CDATA #FIXED "value">
  Example:
  <!ATTLIST tutorial language CDATA #FIXED "EN">

DTD ATTRIBUTE TYPES

- ☞ CDATA is probably the most common attribute type as it allows for plain text to be used for the attribute's value.
- ☞ There may however, be cases where you need to use a different attribute type.
- ☞ When setting attributes for your elements, the attribute TYPE field can be set to one of the following values.

Type	Description
CDATA	Character Data (text that doesn't contain markup)
ENTITY	The name of an entity (which must be declared in the DTD)
ENTITIES	A list of entity names, separated by whitespaces. (All entities must be declared in the DTD)
Enumerated	A list of values. The value of attribute must be one from this list.
ID	A unique ID or name. Must be a valid XML name.
IDREF	Represents the value of an ID attribute of another element.
IDREFS	Represents multiple IDs of elements, separated by whitespace.
NMTOKEN	A valid XML name.
NMTOKENS	A list of valid XML names, separated by whitespace.
NOTATION	A notation name (which must be declared in the DTD).

CDATA

- ☞ As with all attribute types, the attribute type of CDATA is placed after the attribute name and before the default value.

  Syntax:

  <!ATTLIST element_name attribute_name CDATA default_value>

Example: XML

```
<mountains>
        <mountain country="New Zealand">
        <name>Mount Cook </name>
</mountain>
<mountain country="Australia">
<name>Cradle Mountain</name>
</mountain>
</mountains>
```

ENTITY

- ☞ The attribute type of ENTITY is used for referring to the name of an entity you have declared in your DTD.

  Syntax:

<!ATTLIST element_name attribute_name ENTITY default_value>

Example: DTD
<!ATTLIST mountain photo ENTITY #IMPLIED>
<!ENTITY mt_cook_1 SYSTEM "mt_cook1.jpg">

Example : XML
<mountains>
<mountain photo="mt_cook_1">
<name>Mount cook</name>
</mountain>
<mountain>
<name>Cradle Mountain</name>
</mountain>
</mountains>

ENTITIES

- The attribute type of ENTITIES allows you to refer to multiple entity names, separated by a space.

Syntax:
<!ATTLIST element_name attribute_name ENTITIES default_value>

Example: DTD
<!ATTLIST mountain photo ENTITIES #IMPLIED>
<!ENTITY mt_cook_1 SYSTEM "mt_cook1.jpg">
<!ENTITY mt_cook_2 SYSTEM "mt_cook2.jpg">

Example: XML
<mountains>
<mountain photo="mt_cook_1 mt_cook_2">
<name>Mountain Cook </name>
</mountain>
<mountain>
<name>Cradle Mountain</name>
</mountain>
</mountains>

Enumerated

- The enumerated attribute type provides for a list of possible values. This enables the DTD user to provide one value from the list of possible values.

- The values must be surrounded by parentheses and each value must be separated by a pipe (|).

Syntax:

<!ATTLIST element_name attribute_name (value1 | value2 | value3) default_value>

Example: DTD

<!ATTLIST tutorial published (yes | no) "no">

Example: XML

```
<tutorials>
    <tutorial published="yes">
    <name>XML Tutorial </name>
</tutorial>
<tutorial published="no">
    <name>HTML Tutorial</name>
</tutorial>
<tutorial>
    <name>CSS Tutorial</name>
</tutorial>
</tutorials>
```

ID

- The attribute type of ID is used specifically to identify elements.

Note:

Because of this, no two elements can contain the same value for attributes of type ID. Also you can only give an element one attribute of type ID. The value that is assigned to an attribute of type ID must be a valid XML name.

Syntax:

<!ATTLIST element_name attribute_name ID default_value>

Example: DTD

<!ATTLIST mountain mountain_id ID #REQUIRED>

Example: XML

```
<mountains>
    <mountain mountain_id="m10001">
    <name>Mount Cook</name>
    </mountain>
```

```
        <mountain  mountain_id=”m10002”>
        <name>Cradle Mountain</name>
        </mountain>
</mountains>
```

IDREF

- The attribute type of IDREF is used for referring to an ID value of another element in the document.

  Syntax:

  <!ATTLIST element_name attribute_name IDFE default_value>

  Example: DTD

  <!ATTLIST employee employee_id ID #REQUIRED manager_id IDREF #IMPLIED>

  Example: XML

```
<employees>
        <employee  employee_id=”e10001” manager_id=”e10002”>
        <first_name>Ramesh</first_name>
        <last_name>Chaudhary</last_name>
        </employee>
        <employee  employee_id=”e10002”>
        <first_name>Rajan</first_name>
        <last_name>Maharjan</last_name>
        </employee>
</employees>
```

IDREFS

- The attribute type of **IDREFS** is used for referring to the ID values of more than one other element in the document. Each value I separated by a space.

  Syntax:

  <!ATTLIST element_name attribute_name IDREFS default_value>

  Example: DTD

  <!ATTLIST individual individual_id ID #REQUESTED parent_id IDREFS #IMPLIED>

  Example: XML

```
<individuals>
```

```
<individual individual_id =”e10001” parent_id=”e1002 e10003”>
<first_name>Bharat</first_name>
<last_name>Rimal</last_name>
</individual>
<individual individual_id=”e10002”>
<first_name>Mohan</first_name>
<last_name>Shrestha</last_name>
</individual>
<individual individual_id=”e10003”>
<firat_name>Krishna</first_name>
<last_name>Bhandari</last_name>
</individual>
</individuals>
```

NMTOKEN

- ☞ An NMTOKEN (name token) is any mixture of Name characters. It cannot contain whitespace (although leading or trailing whitespace will be trimmed/ignored).
- ☞ While Names have restrictions on the initial character (the first character of a Name cannot include digits, diacritics, the full stop and the hyphen), the NMTOKEN doesn't have these restrictions.

  Syntax:

  <!ATTLIST element_name attribute_name NMTOKEN default_value>

  Example: DTD

  ```
  <!ATTLIST mountain country NMTOKEN #REQUIRED>
  ```

  Example: XML

  ```
  <mountains>
      <mountain country=”NZ”>
      <name>Mount Cook</name>
      </mountain>
      <mountain country=”AU”>
      <name>Cradle Mountain</name>
      </mountain>
  </mountains>
  ```

NMTOKENS

- The attribute type of NMTOKENS allows the attribute value to be made up of multiple NMTOKENS, separated by a space.

  Syntax:

  <!ATTLIST element_name attribute_name NMTOKENS default_value>

  Example: DTD

```
<!ATTLIST mountains country NMTOKENS #RQUIRED>
```

  Example: XML

```
<mountains country=”NZ AU”>
    <mountain>
        <name>Mount Cook</name>
    </mountain>
    <mountain>
        <name>Cradle Mountain</name>
    </mountain>
</mountains>
```

NOTATION

- The attribute type of NOTATION allows you to use a value that has been declared as a notation in the DTD.
- A notation is used to specify the format of non-XML data. A common use of notations is to describe MIME types such as image/ gif, image/jpeg etc.

  Syntax:

  <!NOTATION name SYSTEM “external_id”>

  <!ATTLIST element_name attribute_name NOTATION default_value>

  Example: DTD

```
<!NOTATION GIF SYSTEM “image/gif”>
<!NOTATION JPG SYSTEM “image/jpg”>
<!NOTATION PNG SYSTEM “image/png”>
<!ATTLIST mountain
    Photo ENTITY #IMPLIED
    Photo_type NOTATION (GIF | JPG | PNG) #IMPLIED>
<!ENTITY mt_cook_1 SYSTEM “mt_cook1.jpg”>
```

  Example: XML

```
<mountains>
    <mountain photo=”mt_cook_1” photo_type = “JPG”>
    <name> Mount Cook </name>
    </mountain>
```

```
        <mountain>
            <name>Cradle Mountain</name>
        </mountain>
</moutains>
```

Problem with DTDs

- Written in a language other than XML, so need a separate parser.
- All definitions in a DTD are global, applying to the entire document. Cannot have two elements with the same name but with different content in separate contexts.
- The text content of an element can be PCDATA only, no finer typing for numbers or special string formats.
- Limited typing for attribute values.
- DTDs are not truly aware of namespaces, they recognize prefixes but not the underlying URI.

XML SCHEMA

What is XML Schema?

- XML Schema is an XML based language used to create XML based languages and data models.
- An XML schema defines element and attribute names for a class of XML documents.
- The schema also specifies the structure that those documents must adhere to and the type o content that each element can hold.
- XML documents that attempt to adhere to an XML schema are said to be instances of that schema.
- If they correctly adhere to the schema, then they are valid instances. This is not the same as being well formed. A well formed XML document follows all the syntax rules of XML, but it does not necessarily adhere to any particular schema. So, an XML document can be well formed without being valid, but it cannot be valid unless it is well formed.
- An XML schema describes the structure of an XML instance document by defining what each element must or may contain.

Form of an XML Schema Definition

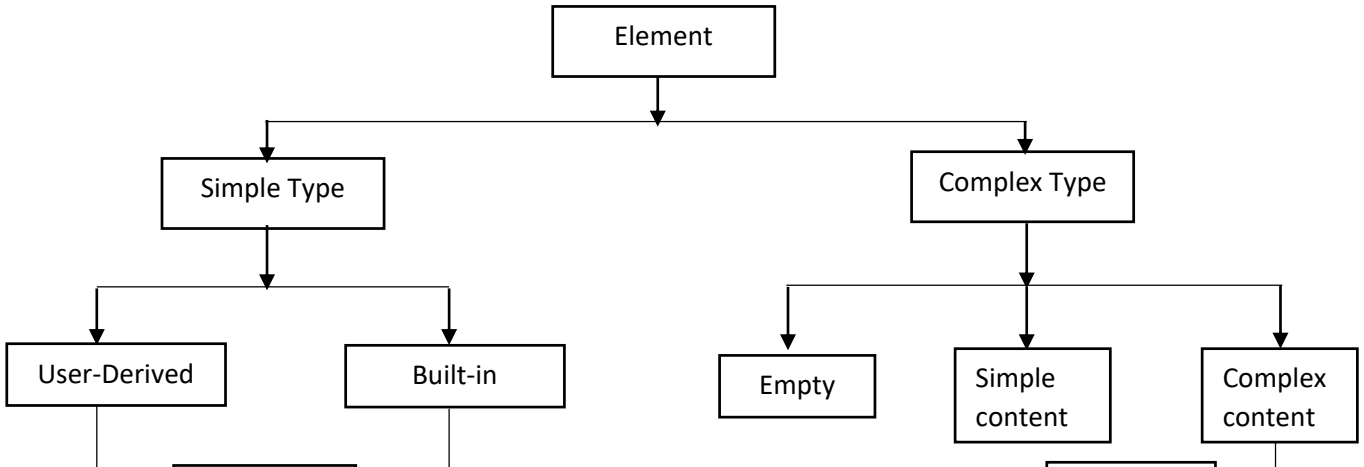
An XML Schema is also an XML document with XSD extension.

First item: xml declaration
<?xml version="1.0" encoding = "UTF-8"?>
XML comments and processing instructions are allowed.

Root element: schema with a namespace declaration.
<xs:schema xmlns:xs = http://www.w3.org/2001/XMLSchema>
</xs:schema>
The possible namespace prefixes are: xs, xsd, or none.

- ☞ The following is a high –level overview of Schema types.
  - ❖ Elements can be of simple type or a complex types.
  - ❖ Simple type elements can only contain text. They cannot have child elements or attributes.
  - ❖ All the built-in types are simple types (e.g. xs:string)
  - ❖ Schema authors can derive simple types by restricting another simple type. For example, an email type could be derived by limiting a string to a specific pattern.
  - ❖ Simple types can be atomic (e.g. strings and integers) or non-atomic (e.g. lists).
  - ❖ Complex-type elements can contain child elements and attributes as well as text.
  - ❖ By default, complex-type elements have complex content, meaning that they have child elements.
  - ❖ Complex-type elements can be limited to having simple content, meaning they only contain text. They are different from simple type elements in that they have attributes.
  - ❖ Complex types can be limited to having no content, meaning they are empty, but they may have attributes.

- ☞ Complex types may have mixed content i.e. a combination of text and child elements.
- ☞ Figure below illustrate the Schema element as:



XML SCHEMA ELEMENT DECLARATION

- Elements are declared using an element named xs: element with an attribute that gives the name of the element being defined.
- Element declarations can be one of two sorts.

Simple Type

- Simple-type elements have no children or attributes. Content of these elements can be text only. A simple-type element is defined using the type attribute.
  Example:
  <xs : element name = “item” type = “xs: string”/>
  <xs : element name = “price” type = “xs : decimal”/>

- The values xs:string and xs:decimal are two different simple types predefined in the XML Schema language.
- XML Schema specifies 44 built in types, 19 of which are primitive.

  The 19 built-in primitive types are listed below:

string	duration	gYear	base64Binary
boolean	dateTime	gMonthDay	anyURI
decimal	time	gDay	QName
float	date	gMonth	NOTATION
double	gYearMonth	hexBinary	

The other 25 built-in data types are derived from one of the primitive types listed above:

Normalized String	ID	negativeInteger	unsigned Int
Token	IDREF	long	unsigned Short

Language	IDREFS	int	unsigned Byte
NMTOKEN	ENTITY	short	positive Integer
NMTOKENS	ENTITIES	byte	
Name	integer	non Negative Integer	
NCName	nonPositiveInteger	unsigned Long	

Code Sample

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

Xml for above code:

```
<Author>
                <FirstName>Ram</FirstName>
                <LastName>Shrestaha</LastName>
</Author>
```

Complex Type

Complex-type elements have attributes, child elements or some combination of both. As the above diagram shows a complex-type element can be empty, contain simple content such as a string or can contain complex content such as a sequence of elements.

Order Indicator: Content Models

- Content models are used to indicate the structure and order in which child elements can appear within their parent element. Content models are made up of model groups. The three types of model groups are listed below:
  1. **xs: sequence** – the elements must appear in the order specified.
  2. **xs:all** – the elements must appear but the order is not important.
  3. **xs:choice** – only one of the elements can appear.

xs: sequence

- ☞ The following sample shows the syntax for declaring a complex-type element as a sequence, meaning that the elements must show up in the order they are declared.
  For example:

```
<xs: element name= “ElementName”>
        <xs: complexType>
            <xs: sequence>
                <xs: element name= “child1” type= “xs:string”/>
                <xs: element name= “child2  type=”xs: string”/>
                <xs: element name= “child3” type= “xs: string”/>
            </xs:sequence>
        <xs: complexType>
</xs: element>
```

xs: all

- ☞ The following sample shows the syntax for declaring a complex-type element as a conjunction, meaning that the elements can show up in any order.

```
<xs: element name= “ElementName”>
        <xs: comlexType>
            <xs: all>
                <xs: element name= “child1” type= “xs: string”/>
                <xs: element name= “child2” type= “xs:string”/>
                <xs: element name = “child3” type= “xs: string”/>
            </xs: all>
        </xs: complexType>
</xs: element>
```

xs: choice

- ☞ The following sample shows the syntax for declaring a complex-type element as a choice, meaning that only one of the child elements may show up.

```
<xs: element name= “ElemnetName”>
        <xs: complexType>
            <xs: choice>
                <xs: element name= “child1” type= “xs: string”/>
                <xs: element name= “child2” type= “xs: string”/>
                <xs: element name= “child3” type= “xs:string”/>
            </xs: choice>
        </xs: complexType>
</xs:element>
```

Example:

```
<xs: element name= “location”>
        <xs: complexType>
        <xs: sequence>
            <xs: element name= “city” type= “xs: string”/>
            <xs: element name= “state” type= “xs: string”/>
        </xs: sequence>
        </xs: complexType>
</xs: element>
```

Note: the element xs: sequence is one of several ways to combine elements in the content.

Corresponding DTD:

```
<!ELEMENT location (city, state)>
```

Corresponding XML:

```
<location>
        <city> Kathmandu </city>
        <state>Province 3 </state>
</location>
```

An xs: element, element may also have attributes that specify the number of occurrences of the element at this position in the sequence.

minOccurs= “0” // default = 1
maxOccurs = “5” // default= maximum (1, minOccurs)
maxOccurs= “unbounded”

Example:
Sample DTD

```
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry ( name, phone, city?)>
<!ELEMENT name (first, middle?,  last)>
<!ELEMENT first  (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)
<!ELEMENT city (#PCDATA)>
```

Convert DTD to XML Schema:

- We use xs: string in place of PCDATA. Element sequencing is handled with xs: sequence. The attributes minOccurs and maxOccurs take care of the * and ? in the DTD.

```
<?xml version= "1.0"?>
<xs: schema xmlns: xs= http://www.w3.org/2001/XMLSchema?>
        <xs: element name= "phoneNumber">
        <xs: complex Type>
        <xs: sequence>
        <xs: element name= "title" type= "xs: string"/>
        <xs: element name = "entities">
        <xs: complex Type>
        <xs: sequence>
        <xs: element name = "entry" min Occurs = "0" maxOccurs= "unbounded">
<xs: complex Type>
<xs: sequence>
        <xs: element name= "name">
        <xs: complex Type>
        <xs: element name= "first" type = "xs: strintg"/>
        <xs: element name= "middle" type = "xs:string" minOccurs= "0"/>
        <xs: element name= "last" type = "xs: string"/>
        </xs: sequence>
        </xs: complex Type>
</xs: element>
        </xs: element name = "phone" type = "xs: string"/>
        </xs: element name = "city" type = "xs: string" minOccurs = "0"/>
        </xs: sequence>
        <xs: complex Type>
        </xs: element>
        </xs: sequence>
        </xs: complex Type>
        </xs: element>
        </xs: schema>
```

CREATE NAMED TYPES

- Define the complex types in the XML Schema definition and give them each a name. These definition will lie at the top level of the schema element.
- The scope of each complex type definition covers the entire schema element (order is of no consequence).

Naming Convention: Use the suffix “Type” when defining a new type in the XML Schema definition.

```
<xs: simpleType name= “salaryType”>
        <xs: restriction base= “xs:decimal”>
            <xs: minInclusive value=”10000”/>
            <xs: maxInclusive value= “90000”/>
        </xs: restriction>
</xs: sipleType>
```

To use this:

```
<xs: element name = “salary” type= “salaryType”/>
```

Example: Named Element:

```
<?xml version= “1.0”?>
<xs:schema  xmlns: xs= “http://ww.w3.org/2001/XMLSchema”>
        <xs: complexType name= “nameType”>
        <xs: sequence>
        <xs:element name= “first” type=”xs:string”/>
        <xs: element name= “middle” type= “xs:string” minOccurs = “0”/>
        <xs: element name = “last”  type=”xs:string”/>
        </xs:sequence>
        </xs: complexType>
        <xs: complexType name = “entryType”>
            <xs: sequence>
                <xs : element name= “name” type= “nameType”/>
                <xs : element name= “phone” type= “xs: string”/>
            <xs : element name = “city”  type = “xs: string” minOccurs = “0”/>
        </xs: sequence>
        </xs : complexType>
        <xs: complexType name = “entriesType”>
            <xs : sequence>
                <xs : element name = “entry”  type = “entryType” minOccurs = “0” maxOccurs = “unbounded”/>
        </xs: sequence>
        </xs: complexType>
        <xs: complexType  name= “phoneType”>
            <xs : sequence>
            <xs: element  name = “title” type = “xs: string”/>
            <xs : element name = “entries” type = “entriesType”/>
            </xs : sequence>
            </xs: complexType>
```

<xs : element name = “phoneNubers” type = “phoneType”/>
</xs : schema>

MIXED CONTENT

- ☞ Mixed content refers to the situation where an element has both text and elements in its content.
- ☞ Because of the sub-elements, the element being defined must be a complex type.
- ☞ To allow mixed content in an element definition, simply add an attribute mixed to the *xs: complexType* starting tag that asserts: **mixed = “true”**

This attribute has a default value of “false”.

XSD:

<xs : element name = “narrative”>
<xs : complexType mixed = “true”>
<xs : choice minOccurs = “0” maxOccurs = “unbounded”>
<xs : element name = “bold” type = “xs : string”/>
<xs : element name = “italics” type = “xs: string”/>
<xs : element name = “underline” type = “xs : string”/>
</xs : choice>
</xs: complexType>
</xs : element>

XML:

<narrative>
Higher beings from <italics> outer space </italics> may not want to tell us the <under line> secrets of life </underline>because we are not ready. But may be they will change their tune after a little <bold>welcome</bold><italics>Rajan Shrestha</italics>
</narrative>

ATTRIBUTE SPECIFICATION

- ☞ Attributes are defined using the element xs: attribute with its own attributes, name and type.
- ☞ The xs: attribute element must lie inside a complex type specification, but the type of the new attribute (its value) must be a simple type.
- ☞ Attribute values may not contain elements of other attributes.

XSD:

<xs : element name= “name”>
<xs: complexType>
<xs: sequence>

```
                    <xs : element name = “first” type= “xs: string”/>
                    <xs : element name= “middle” type = “xs: string” minOccurs = “0”/>
                    <xs : element name = “last” type = “xs: string”/>
                </xs : sequence>
                <xs: attribute name = “gender” type = “xs : string”/>
                </xs : complexType>
</xs : element>
```

XML:

```
<name gender = “male”>
                <first> Ram </first>
                <middle>Kumar</middle>
                <last>Nepali </last>
</name>
```

Where name have gender attribute with value male.

Attributes with Empty Elements

- These specifications are easy. The only content inside the xs:complexType element will be the attribute definitions. This complex type definition can be an anonymous type or a named type.

```
<xs : complexType>
        <xs : attribute name = “src” type = “xs : anyURI”/>
        <xs : attribute name = “width”  type = “xs: integer”/>
        <xs : attribute name = “height” type = “xs: integer”/>
</xs: complexType>
```

Attributes for Elements with Only Text Content

- These specifications are a bit more complicated. We need a complex type to allow for attribute definitions, but we need a simple type to specify the text content.
- The solution is to place an xs: simpleContent element inside of the complex type and use its xs: extension element to specify the simple type.

```
<xs : element   name = “bookTitle”>
        <xs : complexType>
            <xs : simpleContent>
            < xs : extension base = “xs : string”>
            <xs : attribute name = “author”  type = “xs : string”/>
            <xs : attribute name = “isbn” type = “xs : string”/>
            </xs : extension>
        </xs: simpleContent>
```

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

Default and Fixed Value of Attribute:

Default Values

- Attributes can have default values, To specify a default value, use the default attribute of the xs: attribute element.
- Default values for attributes work slightly differently than they do for elements.
- If the attribute is not included in the instance document, the schema processor inserts it with the default value.

For example:

```
<xs : element name = "FirstName">
        <xs : comlexType>
            <xs : simpleContent>
                <xs: extension base = "xs : string">
                <xs : attribute name = "Full" type = "xs : Boolean" default =
"true"/>
                </xs: extension>
                </xs: simpleContent>
                </xs: complexType>
        </xs: element>
```

Fixed Values

- Attribute values can be fixed, meaning that, if they appear in the instance document, they must contain a specified value. Like with simple-type elements, this is done with the fixed attribute.

For example:

```
<xs : element name = "Name">
        <xs : complexType>
            <xs : sequence>
            <xs : element  name = "FirstName">
            <xs : complexType>
            <xs : simpleContent>
            <xs : extension  base = "xs : string">
        <xs : attribute  name = "Full"  type = "xs : Boolean" default = "true"/>
        </xs : extension>
        </xs : simpleContent>
        </xs : complexType>
        </xs : element>
        <xs : element name = "LastName" type = "xs : string"/>
        </xs : sequence>
```

```
<xs : attribute  name = “Ramchandra”  type= “xs : Boolean”  fixed = “true”/>
<xs : attribute name = “HomePage”  type = “xs : anyURI”/>
</xs :coplexType>
</xs : element>
```

Requiring Attributes

- By default, attributes are optional, but they can be required by setting the use attribute of xs : attribute to require as shown below.
  <xs : attribute name = “HomePage” type = “xs : anyURI” use = “required”/>

Restrictions on Content (facets):

- When an XML element or attribute has a data type defined, it puts restrictions on the element 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 attributes are called **facets**.

XSD RESTRICTIONS/ 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.
  For example:

```
<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 define 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>
```

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.

Example:

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

For example:

```
<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:
For example:

```
<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>
```

Restriction on Whitespace Character

- To specify how whitespace characters should be handled, we would use the white space constraint.
- This example defines an element called “address” with a restriction.
- The white space 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>
```

Restriction 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.

For example:

```
<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

Constraints	**Description**
Enumeration	Define a list of acceptable values
Fraction Digits	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.
Max Exclusive	Specifies the super bounds for numeric values (the value must be less than this value)
Max Inclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
Max length	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero.
Min Exclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
Min Inclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
Min Length	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

Total Digits	Specifies the exact number of digits allowed. Must be greater than zero.
White Space	Specifies how white space (line feeds, tabs, spaces and carriage returns) is handled.

XSL (EXTENSIBLE STYLE SHEET LANGUAGE)/XSLT (XSL TRANSFORMATION)

Introduction to XSL and XSLT

- ☞ XSL stands for Extensible Style Sheet Language. It is similar to XML as CSS is to HTML.
- ☞ XSL is a powerful language for applying styles to XML documents.
- ☞ XML also has it's own style language-XSL and is a very powerful language for applying styles to XML documents. XSL has two parts i.e. a formatting language and a transformation language.
- ☞ The formatting language allows you to apply styles similar to what css does. Browser supports for the XSL formatting language is limited at this stage.
- ☞ The transformation language is known as XSLT (XSL Transformations). XSLT allows you to transform your XML document into another form. For example you could use XSLT to dynamically output some (or all) of the contents of your XML file into an HTML document. Containing other content.
- ☞ XSLT enables you to transform XML documents into another form. For example, you can take your XML document, combine it with HTML / CSS and it will look completely different when viewing it in your user browser.

XSLT DOCUMENTS

- ☞ An XSLT document is a valid XML document. An XSLT document consists of a number of elements /tag/ attributes.
- ☞ These can be XSL elements or elements or elements from another language (such as HTML).
- ☞ When you look at an XSLT document, you will notice that it is constructed like any other XML document.

Processing a Transformation

- ☞ A transformation can takes place in one of three locations:
  - ❖ On the server
  - ❖ On the client (e.g. Web browser)
  - ❖ With a standalone program

Need for XSL

- In case of HTML document, tags are predefined such as table, div, and span and the browser knows how to add style to them and display those using CSS styles. But in case of XML documents, tags are not predefined.
- In order to understand and style an XML document, World Wide Web Consortium (W3C) developed XSL which can act as XML based Style sheet Language. A
- An XSL document specifies how a browser should render an XML document.
- The main part of XSL are as follows:
  - **XSLT:** It is used to transform XML documents into various other types of document.
  - **XPath:** It is used to navigate XML document.
  - **XSL-FO:** It is used to format XML document.

XSLT:

- XSLT stands for Extensible Style Sheet Language Transformations, which provides ability to transform XML data from one format to another automatically.

How does XSLT works?

An XSLT style sheet is used to define the transformation rules to be applied on the target XML document. XSLT style sheet is written in XML format. XSLT Processor takes the XSLT style sheet and applies the transformation rules on the target XML document and then it generates a formatted document in the form of XML, HTML or text format. This formatted document is then utilized by XSLT formatter to generate the actual output which is to be displayed to the end-user.

XSLT <template> Element

- <xsl: template> defines a way to reuse templates in order to generate the desired output for nodes of a particular type / context.

  Eg.

  <xsl: template
  Name = Qname
  Match = Pattern
  Priority = number
  Mode = QName>
  </xsl:template>

  Where,

  Name: Name of the element on which template is to be applied.

  Match: Pattern which signifies the elements on which template is to be applied.

  Priority: Priority number of a template

Mode: Allows element to be processed multiple times to produce a different result each time.

XSLT<value-of>

- <xsl:value-of> tag puts the value of the selected node as per XPath expression, as text.
  Eg.
  <xsl:value-of select=expression disable-output-escaping = "yes" | "no">

  </xsl:value-of>

  Attributes
  Select: XPath Expression to be evaluated in current context.
  Disable-output escaping: Default- "no". If "yes", output text will not escape xml characters from text.

XSLT <for-each>

<xsl: for-each> tag applies a template repeatedly for each node.
Eg.
<xsl: for-each select = expression>
</xsl: for-each>

Attributes
Select: XPath Expression to be evaluated in the current context to determine the set of nodes to be iterated.

XSLT <sort>

- This tag specifies a sort criteria on the nodes.
  <xsl: sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" | QName }
  order = { "ascending" | "descending" }
  case- order = { "upper-first" | "lower-first" }>
  </xsl: sort>

Attributes
Select: Sorting key of the node.
Lang: Language alphabet used to determine the sort order.
Data-type: Data type of the text.
Order: Sorting order. Default is "ascending".
Case-order: Sorting order of string by capitalization. Default is "upper-first".

<xsl : if> Element

- This tag specifies a conditional test against the content of nodes.
  <xsl: if
  test = Boolean-expression>
  </xsl: if>

Example: <xsl: if test = "marks > 90">
<xsl : choose> Element
<xsl : choose> tag specifies a multiple conditional tests against the content of nodes in conjunction with the <xsl: otherwise> and <xsl: when> elements.

```
<xsl : choose>
        <xsl : when test = "marks > 90">
            High
        </xsl : when>
        <xsl : when test = "marks > 85">
            Medium
        </xsl : when>
        <xsl : otherwise>
            Low
        </xsl: otherwise>
</xsl: choose>
```

XQUERY

Introduction to XQUERY:

- XQUERY is a functional query language used to retrieve information stored in XML format. It is same as for XML what SQL is for databases.
- It was designed to query XML and XQURERY is built on XPath expressions. It is a W3C recommendation which is supported by all major databases.
- The definition of XQurery given by its official documentation is as follows:
- It is standardized language for combining documents, databases, webpages and almost anything else. It is very widely implemented. It is powerful and easy to learn. XQuery is replacing proprietary middleware languages and Web Application development languages. XQuery is replacing complex Java or C^{++} programs with a few lines of code. XQuery is simpler to work with and easier to maintain than may other alternatives.

Use of XQuery

- XQuery is a functional language which is responsible for finding and extracting elements and attributes from XML documents. Some use of XQuery are as follows:
  - To extract information to use in a web service.

- ❖ To generates summary reports.
- ❖ To transform XML data to XHTML.
- ❖ Search Web documents for relevant information.

XQurery Features

- ☞ There are many features of XQuery query language. Some important features are as follows:
  - ❖ XQuery is a functional language. It is used to retrieve and query XML based data.
  - ❖ XQuery is expression-oriented programming language with a simple type system.
  - ❖ XQuery is analogous to SQL. For example: As SQL is query language for databases, same as XQuery is query language for XML.
  - ❖ XQuery is XPath based and uses XPath expressions to navigate through XML documents.
  - ❖ XQuery is a W3C standard and universally supported by all major databases.

Advantages of XQuery

- ❖ XQuery can be used to retrieve both hierarchal and tabular data.
- ❖ XQuery can also be used to query tree and graphical structures.
- ❖ XQuery can used to build Web pages.
- ❖ XQuery can be used to query Web pages.
- ❖ XQuery is best for XML based databases and object-based databases. Object databases are much more flexible and powerful than purely tabular databases.
- ❖ XQuery can be used to transform XML documents into XHTML document.

XQuery Example

Course XML File

```
<?xml version = “1.0” encoding = “UTF-8” ?>
<courses>
        <course category = “JAVA”>
        <title lang = “en”>Learn Java in 3 Months.</title>
        <trainer>Hemant Sharma</trainer>
        <year>2022</year>
        <fees>25000</fees>
        </course>
        <course category = “Dot Net”>
        <title lang = “en”>Learn Dot Net in 3 Months</title>
        <trainer>Roshan Bhusal</trainer>
        <year>2021</year>
        <fees>20000</fees>
```

```
        </course>
        <course category = "C">
        <title lang ="en">Learn C programming in 2 months</title>
        <trainer>Ramesh Kumar</trainer>
        <year>2020</year>
        <fees>5000</fees>
        </course>
        <course category= "XML">
        <title lang = "en">Learn XML in 2 Months. </title>
        <trainer>Ajeet Kumar</trainer>
        <year>2019</year>
        <fees>4000</fees>
        </course>
</courses>
```

Xquery file courses.xqy

```
For $x in doc("courses.xml")/courses/course
Where $x/fees>5000
Return $x/title
```

XPATH (XML Path Language)

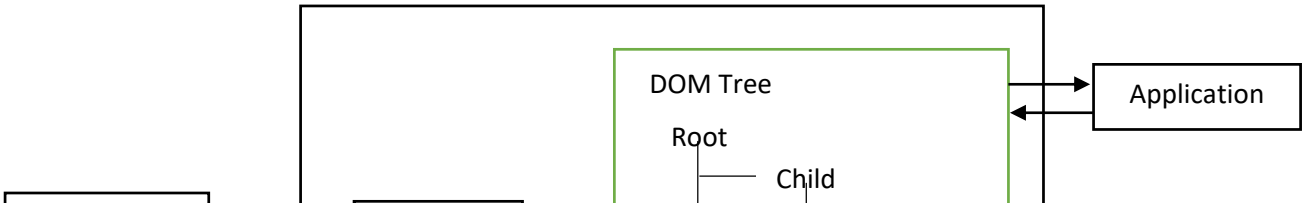
- ☞ XPth is a language that describes a way to locate and process items in Extensible Markup Language (XML) documents by using an addressing syntax based on a path through the document's logical structure or hierarchy.
- ☞ This makes writing programming expressions easier than if each expression had to understand typical XML markup and its sequence in a document.
- ☞ It also allows the programmer to deal with the document at a higher level of abstraction.
- ☞ XPath is a language that is used by and specified as part of both the Extensible Stylesheet language Transformations (XSLT) and XPointer (SML Pointer Language).
- ☞ It uses the information abstraction defined in the XML Information Set (Infoset). Since XPath does not use XML syntax itself, it could be used in contexts other than those of XML.
- ☞ Some important point remember about XPath as:
    - ❖ XPath stands for XML Path Language
    - ❖ XPath uses "path like" syntax to identify and navigate nodes in an XML document.
    - ❖ XPath conteains over 200 built-in functions
    - ❖ XPath is a major element in the XSLT standard
    - ❖ XPath is a W3C recommendation

SAX (Simple API for XML)

- SAX (Simple API for XML) is an application program interface (API) that allows a programmer to interpret a Web file that uses the Extensible Markup Language (XML), i.e. a Web file that describes a collection of data.
- SAX is an alternative to using the Document Object Model (DOM) to interpret the XML file.
- As its name suggests, it's a simpler interface than DOM and is appropriate where many or very large files are to be processed, but it contains fewer capabilities for manipulating the data content.
- SAX is an event-driven interface. The programmer specifies an event that may happen and if it does, SAX gets control and handles the situation. SAX works directly with an XML parser.

DOM (Document Object Model)

- The DOM is an application programming interface (API) for HTML and XML documents.
- It defines the logical structure of documents and the way a document is accessed and manipulated.
- In the DOM specification, the term "document" is used in the broad sense – increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents.
- With the help of DOM, programmers can build documents, navigate their structure, and add, modify or delete elements and content.
- Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions – in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.
- As a W3C specification, one important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and applications.
- The XML DOM is a standard object model for XML, XML documents have a hierarchy of informational units called nodes, DOM is a standard programming interface of describing those nodes and the relationships between them.
- As XML DOM also provides an API that allows a developer to add, edit, move or remove nodes at any point on the tree in order to create an application.
- Figure below illustrate the diagram for the DOM structure as:



Advantage of XML DOM

- The advantages of XML DOM are as follows:
  - XML DOM is language and platform independent.
  - XML DOM is traversable i.e. information in XML DOM is organized in a hierarchy which allows developer to navigate around the hierarchy looking for specific information.
  - XML DOM is modifiable i.e. It is dynamic in nature providing the developer a scope to add, edit, move or remove nodes at any point on the tree.

Disadvantages of XML DOM

- The disadvantages of XML DOM are as follows:
  - It consumes more memory (if the XML structure is large) as program written once remains in memory all the time until and unless removed explicitly.
  - Due to the extensive usage of memory, its operational speed, compared to SAX is slower.

XML DOM – Model

- A DOM document is a collection of nodes or pieces of information, organized in a hierarchy, some types of nodes may have child nodes of various types and others are leaf nodes that cannot have anything under them in the document structure
- Following is a list of the node types with a list of node types that they may have as children.
  - **Document:** Element (maximum of one), Processing Instruction, Comment, Document Type (maximum of one)

- **Document Fragment:** Element, Processing Instruction, Comment, Text, CDATA Section, Entity Reference.
- **Entity Reference:** Element, Processing Instruction, Comment, Text, CDATA Section, Entity Reference.
- **Element:** Element, Text, Comment, Processing Instruction, CDATA Section, Entity Reference.
- **Attr:** Text, Entity Reference
- **Processing Instruction:** No children
- **Text:** No children
- **CDATA Section:** No children
- **Entity:** Element, Processing Instruction, Comment, Text, CDATA Section, Entity Reference.
- **Notation:** No children

Example:
Consider the DOM representation of the following XML document **node.xml**

```
<?xml version = "1.0" ?>
<Company>
<Employee category = "technical">
<FirstName>Mohan</FirstName>
<LastName>Shrestha</LastName>
<ContactNo>9841983339</ContactNo>
</Employee>
<Employee category= "non-technical">
<FirstName>Taniya</FirstName>
<LastName>Mishra</LastName>
<ContactNo>9805623012</ContactNo>
</Employee>
</Company>
```

Creating XML Parser

- The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.
- However, before an XML document can be accessed, it must be loaded into an XML DOM object.
- All modern browsers have a built-in XML parser that can convert text into an XML DOM object

Parsing a Text String

- This example parses a text string into an XML DOM object and extracts the info from it with JavaScript.

For example:

```
<html>
```

```
<body>
<p  id = “demo”></p>
<script>
Var   text,  parser, xmlDoc;
Text = “<bookstore><book>” +
“<title>Everyday Italian</title>” +
“<author>Yeshbant Singh</author>” +
“<year>2010</year>” +
“</book></bookstore>”;
Parser = new DOMParser();
xmlDoc = parser.parseFromString( text, “text/xml”);
document.getElementById(“demo”).InnerHTML =
xmlDoc.getElementsByTagName(“title”) [0].childNodes[0].nodeValue;
</script>
</body>
</html>
```