Unit 3: The Client Tier (10 Hrs.)

Contents

Representing content

Introduction to XML

XML stands for extended markup language and was designed to store and transport data.XML is a markup language 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.XML 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 carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

example:

```
<note>
    <date>2016-06-09</date>
    <to>Anil</to>
    <from>Anju</from>
    <heading>Reminder</heading>
    <body>Don't forget to wish</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>, <hl>, etc.). In contrast, XML allows the author to define his/her own tags and his/her own document structure. The XML processor cannot 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>Anil</to>
        <from>Anju</from>
         <heading>Reminder</heading>
        <body>Don't forget to bunk the web tech class at Biratnagar!</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>Anil</to>
<from>Anju</from>
<heading>Reminder</heading>
<body>Don't forget to bunk the web tech class at Biratnagar!</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 Anil from Anju.

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 rule:

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

All XML Elements Must Have a Closing Tag: In XML, it is illegal to omit the closing tag. All elements must have a closing tag: eg:<message></message>

XML tags are case sensitive: The tag <Message> is different from the tag <message>. Opening and closing tags must be written with the same case.

XML Elements Must be Properly Nested: In XML, all elements must be properly nested within each other for example

<note>
<to>.....</to>
<from>.....</from>
</note>

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>
```

XML Attribute Values Must be Quoted: In XML, the attribute values must always be quoted for example <gender="male">

```
<note date=06/01/2012>
  <to>Anil</to>
  <from>Yadav</from>
</note>
```

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 **<** 1000 then</message>

There are 5 predefined entity references in XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Comments in XML: comment in XML is similar to HTML .example < !-- This is a comment -->

White-space is preserved in XML: The white-space in a document is not truncated

XML Usage

XML is used in many aspects of web development. It is often used to separate data from presentation.

Here are some lists of XML usages:

- XML Separates Data from HTML
- XML Simplifies Data Sharing
- XML Simplifies Data Transport
- XML Simplifies Platform Changes
- XML Makes Your Data More Available
- XML Used to Create New Internet Languages like XHTML, WSDL, WAP and WML, SMIL etc.

Elements and Attributes

Elements

XML elements are everything including start tag to end tag.

An element can contain:

- other elements
- text

- attributes
- or a mix of all of the above…

syntax:

```
<element-name att1 att2>
        ....content
</element-name>
```

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>
```

Empty elements

An empty element (element with no content) has following syntax <*element-name att1 att2…/*>

XML elements Naming Rules

XML elements must follow these naming rules:

- Element names are case-sensitive. For example, Address, address, and ADDRESS are different names.
- Element names must start with a letter or underscore.
- Element names cannot start with the letters xml (or XML, or Xml, etc)
- Element names can contain letters, digits, hyphens, underscores, and periods.
- Element names cannot contain spaces.

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 eoa are perfectly legal in XML, but watch out for problems if your software vendor doesn't support them.

Attributes

XML elements can have attributes, just like HTML. Attributes provide additional information about an element. In HTML, attributes provide additional information about elements. An XML attribute is always a name-value pair.

syntax:

<element-name *att1 att2* >
....content..
< /element-name>

Where *att1* and *att2* has the following form:

name = "value"
value has to be in double (" ") or single (' ') quotes. Here, *att1* and *att2* are unique attribute labels.

Example:

<person gender='female'>

XML elements vs Attributes

Lets look these examples:

```
<person gender="male">
    <firstname>Anil</firstname>
    <lastname>Yadav</lastname>
</person>
<person>
    <gender>male</gender>
    <firstname>Anil</firstname>
    <lastname>Yadav</lastname>
</person>
```

In the first example, gender is an attribute. in second ,gender is element.both examples provides the same information. There are no rules about when use attributes or when to use elements.

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>Anil</to>
  <from>Anju</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Biratnagar!</body>
</note>

<note id="502">
  <to>Anju</to>
  <from>Anil</from>
  <heading>Re: Reminder</heading>
  <body>Ok Anju !!</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 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 an example:
This XML carries HTML table info:

```
<table>
    <tr>
        <td>Apple </td>
        <td>ball</td>
        <td>cat</td>
    </tr>
<table>
```

again, this XML carries furniture table info:

```
<table>
    <name>Coffee table</name>
    <length>100</length>
    <width>50</width>
</table>
```

If above two fragments were added together, there would be a name conflict. An XML parser will not know how to handle these conflicts. Thus, *xmlns* tagged XML namespace are used to provide unique name to element and attributes in an XML document. *xmlns* are defined in W3C documentation. The *xmlns* 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>Apple</h:td>
  <h:td>Ball</h:td>
  <h:td>Cat</h:td>
 </h:tr>
</h:table>

<f:table>
 <f:name> Coffee Table</f:name>
 <f:width>100</f:width>
 <f:length>50</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>Apple</h:td>
  <h:td>Ball</h:td>
  <h:td>Cat</h:td>
 </h:tr>
</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">
 <f:name> Coffee Table</f:name>
 <f:width>100</f:width>
 <f:length>50</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="https://www.w3schools.com/furniture">
```

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**. 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. (Wikipedia)

Things to remember

- XML stands for extended markup language and was designed to store and transport data.XML is a markup language much like HTML.
- XML tags are not predefined. You must define your own tags.
- The syntax rules of XML are: All XML Elements Must Have a Closing Tag, XML tags are case sensitive, XML Elements Must be Properly Nested.XML Documents Must Have a Root Element, XML Attribute Values Must be Quoted, Entity Reference, Whitespace is preserved in XML
- XML elements are everything including start tag to end tag .element can contain other elements, text, attributes or mix of all
- Attributes provide additional information about an element. An XML attribute is always a name-value pair
- XML Namespaces provide a method to avoid element name conflicts.
- 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.

XML Schema

XML Schema is an XML schema language which is an 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

syntax:
You need to declare a schema in your XML document as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

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.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="studentInfo">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="name" type="xs:string" />
                                <xs:element name="address" type="xs:string" />
                                <xs:element name="phone" type="xs:int" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

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

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 (studentInfo,name,address,phone.) 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="https://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://www.w3schools.com note.xsd">

<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
```

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

The following fragment:

```
xmlns="https://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 "https://www.w3schools.com" namespace.

Once you have the XML Schema Instance namespace available:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

you can use the schemaLocation attribute. This attribute has two values, separated by a space. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace:

```
xsi:schemaLocation="https://www.w3schools.com note.xsd"
```

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="https://www.w3schools.com"
xmlns="https://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>
```

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.

Simple type element

Simple type element is used only in the context of the text.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.

Some of the predefined simple types are as follows:

- xs:string
- xs:decimal

- xs:integer
- xs:boolean
- xs:date
- xs:time

Example

Here are some XML elements:

```
<lastname>Anil</lastname>
<age>38</age>
<dateborn>1978-03-27</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 value 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:integer" and contains a string like "Hey there", the element will not validate.

With XML Schemas, you can also add your own restrictions to your XML elements and attributes.

These restrictions are called facets *(restrictions are used to define acceptable values for XML element or attributes.)*

XSD Restrictions/ Facets

Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.

Here we are discussing different types of restriction they are as follows:

Restriction 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>
```

Restriction 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.

Use of pattern Restrictions

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

Restrictions on a Series of Values

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

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

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

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

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

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

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

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

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

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

Restrictions on Whitespace Characters

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

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

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

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

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

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

```
<xs:element name="address">
 <xs:simpleType>
  <xs:restriction base="xs:string">
```

```
    <xs:whiteSpace value="collapse"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

Restrictions on Length

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

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

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

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

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

Restriction on 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

Complex type element

A complex element is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

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

Example of the complex elements:

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

```
<product pid="1345"/>
```

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

```
< student>
 <firstname>Anil</firstname>
 <lastname>Yadav</lastname>
 <lastname>Biratnagar</lastname>
</student>
```

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

```
<game type="outdoor">cricket</game>
```

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

```
<description>
It happened on <date lang="norwegian">03.01.1012</date>
....
</description>
```

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

lets look an example:

```
< student>
 <firstname>Anil</firstname>
 <lastname>Yadav</lastname>
 <lastname>Biratnagar</lastname>
</student>
```

The "student" element can be declared directly by naming the element, like this:

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

If you use the method described above, only the "student" element can use the specified complex type. Note that the child elements, "firstname" , "lastname",and "address" are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared.
The "student" element can have a type attribute that refers to the name of the complex type to use

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

Order indicators

They are used to define the order of the elements. They contain:

1. All indicator
2. Choice indicator
3. Sequence indicator

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

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

Occurrence indicators

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

1. maxOccurs
2. minOccurs

maxOccurs indicator

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

```
<xs:element name="student">
<xs:complexType>
    <xs:sequence>
        <xs:element name="full_name" type="xs:string"/>
        <xs:element name="subject_pass" type="xs:string" maxOccurs="5"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
```

minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

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

The example above indicates that the "subject_pass" element can occur a minimum of zero times and a maximum of five times in the "student" element.

Things to remember

- XML Schema is an XML schema language which is an alternative to DTD.it supports namespaces and data types.You need to declare a schema in your XML document as follows:
- The element is the root element of every XML Schema.
- Simple type element is used only in the context of the text.The syntax for defining a simple element is: where xxx is the name of the element and yyy is the data type of the element.
- A complex element is an XML element that contains other elements and/or attributes.There are four kinds of complex elements:
    1. empty elements
    2. elements that contain only other elements
    3. elements that contain only text
    4. elements that contain both other elements and text
- They are used to define the order of the elements.They contain:
    1. All indicator
    2. Choice indicator
    3. Sequence indicator
- The indicator specifies that the child elements can appear in any order, and that each child element must occur only once
- The indicator specifies that either one child element or another can occur:

- The indicator specifies that the child elements must appear in a specific order:
- Occurrence indicators are used to define how often an element can occur.They contain:
    1. maxOccurs
    2. minOccurs
- The indicator specifies the maximum number of times an element can occur
- The indicator specifies the minimum number of times an element can occur

Document Type Definition (DTD)

Introduction

A DTD defines the structure of XML documents. 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.

DTD is a provide a framework for validating XML documents. You can create DTD files that are shareable to a different application.In XML you can define tags without defining what tag are legal. But defined XML document structure must conform if you specify DTD rules. 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.

You can specify DTD either internally within XML document or externally.

The internal DTD

You can write rules inside XML document using <!DOCTYPE ... > declaration.
Scope of this DTD within this document. Advantages is document validated by itself without external reference.

syntax:

<!DOCTYPE root-element [element-declarations]>

where root-element is the name of root element and element-declarations is where you declare the elements.

Example XML document with an internal DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE studentInfo [
<!ELEMENT studentInfo (name,address,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT adderss (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
]>
<studentInfo>
<name>Anil</name>
<address>Biratnagar</address>
<phone>984111111</phone>
</studentInfo>
```

The DTD above is interpreted like this:

- **!DOCTYPE studentInfo** defines that the root element of this document.

- **!ELEMENT studentInfo** defines that the studentInfo element contains three elements: "name,address,phone"
- **!ELEMENT name** defines the name element to be of type "#PCDATA"
- **!ELEMENT address** defines the address element to be of type "#PCDATA"
- **!ELEMENT phone** defines the phone element to be of type "#PCDATA"

The External DTD:

External DTDs are used to create a common DTD that can be shared between multiple documents. Any changes that are made to the external DTD automatically updates all the documents that reference it. There are two types of external DTDs: (wattle software)

1. private external DTD
2. public external DTD

Private External DTD:

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

syntax of private external DTD:

```
<!DOCTYPE root-element SYSTEM "DTD_location">.
```

where DTD_location is relative or absolute URL.

example:

```
<?xml version="1.0"?>
<!DOCTYPE studentInfo SYSTEM "studentInfo.dtd">
<studentInfo>
<name>Anil</name>
<address>kathmandu</adderss>
<phone>984111111</phone>
</studentInfo>
```

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

```
<!ELEMENT studentInfo (name, address, phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

public External DTD:

public external DTD are identified by the keyword PUBLIC and are intended for broad use.

syntax:

```
<!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">
```

The "DTD_location" is used to find the public DTD if it cannot be located by the "DTD_name".
syntax for writing DTD_name:

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

Example:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

The following prefix 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 Element

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.

syntax :

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

For example,

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

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

syntax :

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

For example,

```
<!ELEMENT address (#PCDATA)>.
```

Elements with any content are declared with the category keyword ANY, can contain any combination of parsable data.

syntax :

```
<!ELEMENT element-name ANY>.
```

For example,

```
<!ELEMENT studentInfo ANY>.
```

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

syntax:

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

For example,

```
<!ELEMENT studentInfo (name,address,phone)>.
```

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.

syntax:

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

For example,

```
<!ELEMENT studentInfo (name)>.
```

This example declares that the child element "name" must occur once, and only once inside the "studentInfo" element.

We can also declare minimum one occurrence of an element.

syntax:

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

For example,

```
<!ELEMENT studentInfo (phone+)>.
```

The + sign in the example above declares that the child element "phone" must occur one or more times inside the "studentInfo" element.

We can declare mixed content.
syntax:

<!ELEMENT element-name (#PCDATA|child1|child2|child3)*>

For example,

<!ELEMENT StudentInfo (#PCDATA|name|address|phone)*>.

This example declares that the "studentInfo" element can contain zero or more occurrences of parsed character data, "name", "address", or "phone" elements.

Defining Attributes

In a DTD, attributes are declared with an ATTLIST declaration.
syntax:
<!ATTLIST element-name attribute-name attribute-type default-value>
for example:
<!ATTLIST person gender CDATA "male">

And its XML example is
<person gender="male" />

As you can see from the syntax above, the ATTLIST declaration defines the element which can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value.
The **attribute-type** can have the following values:(w3school)

Value	Explanation
CDATA	The value is character data(text that doesn't contain markup)
(en1|en2|..)	The value must be an enumerated value
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 value is 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 predefined xml value

The **attribute-default-value** can have the following values:

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 attribute

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 attribute

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 attribute

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>
]>
```

Things to remember

- A DTD defines the structure of XML documents.It is an XML schema language whose purpose is to define legal building blocks of an XML document.
- You can specify DTD either internally within XML document or externally.
- You can write rules inside XML document using declaration. Scope of this DTD within this document.syntax:
- External DTDs are used to create a common DTD that can be shared between multiple documents.There are two types of external DTDs
    1. private external DTD
    2. public external DTD
- Private external DTDs are identified by the keyword SYSTEM, and are intended for use by a single author or group of authors.syntax:.
- public external DTD are identified by the keyword PUBLIC and are intended for broad use.syntax:
- In a DTD, elements are declared with an ELEMENT declaration with the following syntax.<!ELEMENT element-name category>Or<!ELEMENT element-name (element-content)>
- In a DTD, attributes are declared with an ATTLIST declaration.syntax:

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

XSL Language

XSL stands for **EXtensible Stylesheet Language**. The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Stylesheet Languages. It describes how the XML document should be displayed. It consists of four parts :

1. XSLT
2. XPath
3. XSL-FO
4. XQuery

XSLT:

XSLT stands for **Extensible Stylesheet Language Transformations**. It is the most important part of XSL. XSLT transforms an XML document into another XML document. XSLT uses XPath to navigate in XML documents. XSLT does this by transforming each XML element into an (X) HTML element. With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more. XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

Stylesheet Declaration

The root element that declares the document to be an XSL style sheet is ***<xsl:stylesheet>*** or ***<xsl:transform>.***
<xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used.
The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

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

or:

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

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

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

Consider an example below, which we want to transform the following XML document into XHTML : consider XML document

Start with a Raw XML Document - cdcatalog.xml

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

```
 <cd>
  <title>Empire Burlesque</title>
  <artist>Bob Dylan</artist>
  <country>USA</country>
  <company>Columbia</company>
  <price>10.90</price>
  <year>1985</year>
 </cd>
.
.
</catalog>
```

Create an XSL Style Sheet

We can create an XSL Style Sheet ("**cdcatalog.xsl**") with a transformation template:

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

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

<xsl:template match="/">
 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
  <tr bgcolor="#9acd32">
   <th>Title</th>
   <th>Artist</th>
  </tr>
  <xsl:for-each select="catalog/cd">
  <tr>
   <td><xsl:value-of select="title"/></td>
   <td><xsl:value-of select="artist"/></td>
  </tr>
  </xsl:for-each>
 </table>
 </body>
 </html>
</xsl:template>

</xsl:stylesheet>
```

Link the XSL Style Sheet to the XML Document

Now link the XSL Style Sheet to the XML Document. For this, add the XSL style sheet reference to the above mentioned XML document ("**cdcatalog.xml**") as:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
 <cd>
  <title>Empire Burlesque</title>
```

```
  <artist>Bob Dylan</artist>
  <country>USA</country>
  <company>Columbia</company>
  <price>10.90</price>
  <year>1985</year>
 </cd>
.
.
</catalog>
```

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

XSLT <xsl:template> Element:
An XSL style sheet consists of one or more set of rules that are called templates. The ***<xsl:template>*** element is used to build templates.

The **match** attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

Consider the example of cdcatalog.xsl, discussed above, it can be explained as;

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
  <tr bgcolor="#9acd32">
   <th>Title</th>
   <th>Artist</th>
  </tr>
  <tr>
   <td>.</td>
   <td>.</td>
  </tr>
 </table>
 </body>
 </html>
</xsl:template>

</xsl:stylesheet>
```

Since an XSL style sheet is an XML document, it always begins with the XML declaration: **<?xml version="1.0" encoding="UTF-8"?>**.

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

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

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

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

XSLT <xsl:value-of > Element:

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

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
  <tr bgcolor="#9acd32">
   <th>Title</th>
   <th>Artist</th>
  </tr>
  <tr>
   <td><xsl:value-of select="catalog/cd/title"/></td>
   <td><xsl:value-of select="catalog/cd/artist"/></td>
  </tr>
 </table>
 </body>
 </html>
</xsl:template>

</xsl:stylesheet>
```

In the above example cdcatalog.xsl, we have used it as; <xsl:value-of select="catalog/cd/title"/> <xsl:value-of select="catalog/cd/artist"/>

The **select** attribute, in the example above, contains an XPath expression. An XPath expression works like navigating a file system; a forward slash (/) selects subdirectories.

XSLT <xsl:for-each> Element:

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

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

```
<xsl:for-each select="catalog/cd">
<tr>
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
```

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

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

Legal filter operators are:

- = (equal)
- != (not equal)
- < less than
- > greater than

In the above example of cdcatalog.xsl; we can use restriction as;

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

XSLT <xsl:sort> Element:

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

```
<xsl:for-each select="catalog/cd">
<xsl:sort select="artist"/>
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
```

XSLT <xsl:if> Element:

The <xsl:if> element is used to put a conditional test against the content of the XML file. To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

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

In above example cdcatalog.xsl, we can write it as:

```
<xsl:for-each select="catalog/cd">
<xsl:if test="price &gt; 10">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:if>
</xsl:for-each>
```

XSLT <xsl:choose> Element:

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

syntax :

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

In the above example of cdcatalog.xsl, we can embed this element as;

```
<xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<xsl:choose>
<xsl:when test="price &gt; 10">
<td bgcolor="#ffooff"><xsl:value-of select="artist"/></td>
</xsl:when>
<xsl:otherwise>
<td><xsl:value-of select="artist"/></td>
</xsl:otherwise>
</xsl:choose>
</tr>
</xsl:for-each>
```

We can put sequence of the <xsl:when> as;

```
<xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<xsl:choose>
<xsl:when test="price &gt; 10">
<td bgcolor="#ffooff">
<xsl:value-of select="artist"/></td>
</xsl:when>
<xsl:when test="price &gt; 9">
<td bgcolor="#cccccc"> <xsl:value-of select="artist"/></td>
</xsl:when>
<xsl:otherwise>
<td><xsl:value-of select="artist"/></td>
</xsl:otherwise>
</xsl:choose>
</tr>
</xsl:for-each>
```

XPath

XPath is a language and is used for finding the information in an XML document by using an addressing syntax based on a path through the document's logical structure or hierarchy. It is a language for

addressing part of document ,designed to be used by both XSLT and XPointer. XPath contains a library of standard functions. It is a major element in XSLT.In Xpath there are seven different kinds of nodes they are :element, attribute,text, namespace, processing-instruction, comment and document nodes.XML documents are treated as tree of nodes .the topmost element of the tree is called the root element.XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps.
We will use the following XML document in the examples below.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
 <title lang="en">Harry Potter</title>
 <price>29.99</price>
</book>
<book>
 <title lang="en">Learning XML</title>
 <price>39.95</price>
</book>
</bookstore>
```

Selecting Nodes:
XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

Predicates:

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets. In the table below we have listed some path expressions with predicates and the result of the expressions:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element.
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
//title[@lang='eng']	Selects all the title elements that have an attribute named lang with a value of 'eng'

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML elements.

wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

Selecting Several Paths:

By using the | operator in an XPath expression you can select several paths. In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document
/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

XQuery:

XQuery is a functional, expression-oriented language which is designed to query XML data not just a XML file, but anything that can appear as XML, including database. XQuery contains a superset of XPath expression syntax to address specific parts of an XML document.

It uses "FLWOR expression" for performing Operation. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN.

Some basic syntax rules:

- XQuery is case-sensitive
- XQuery elements, attributes, and variables must be valid XML names
- An XQuery string value can be in single or double quotes
- An XQuery variable is defined with a $ followed by a name, e.g. $firstname
- XQuery comments are delimited by (: and :), e.g. (: this is XQuery Comment :)

Consider a XML example;

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
```

```
  <price>29.99</price>
</book>

<book >
…..
</book>
</bookstore>
```

XQuery to extract the data can be written as in following example;

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
return $x/title
```

SAX:

SAX, which stands for **Simple API for XML**. It is an event-based sequential access parser API developed by the XML-DEV mailing list for XML documents.

It provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially. It is used for state-independent processing of XML documents, in contrast to StAX (Streaming API for XML) that processes the documents state-dependently.

SAX parsers have certain benefits over DOM-style parsers. The quantity of memory that a SAX parser must use in order to function is typically much smaller than **that** of a DOM parser. DOM parsers must have the entire tree in memory before any processing can begin, so the amount of memory used by a DOM parser depends entirely on the size of the input data. The memory footprint of a SAX parser, by contrast, is based only on the maximum depth of the XML file (the maximum depth of the XML tree) and the maximum data stored in XML attributes on a single XML element. Both of these are always smaller than the size of the parsed tree itself.

Because of the event-driven nature of SAX, processing documents can often be faster than DOM-style parsers. Memory allocation takes time, so the larger memory footprint of the DOM is also a performance issue. Due to the nature of DOM, streamed reading from disk is impossible. Processing XML documents larger than main memory is also impossible with DOM parsers, but can be done with SAX parsers. However, DOM parsers may make use of disk space as memory to sidestep this limitation.(Wikipedia)

DOM:

DOM stands for Document Objet Model. It is an application programming interface (API) for HTML and XML documents. The Document Object Model is a platform- independent and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The DOM views document as a tree-structure. All elements can be accessed through the DOM tree. Their content (text and attributes) can be modified or deleted, and new elements can be created. The elements, their text, and their attributes are all known as nodes. The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree.

A DOM example using HTML (computerhope)

Consider the following HTML document:

```
<html>
<head>
<title>Example</title>
</head>
<body>
```

```
<h1>Example Page</h1>
<p>This is an example page.</p>
</body>
</html>
```

The DOM for this document includes all of the elements and any text nodes within those elements. The code in the previous example creates an object hierarchy as shown below.

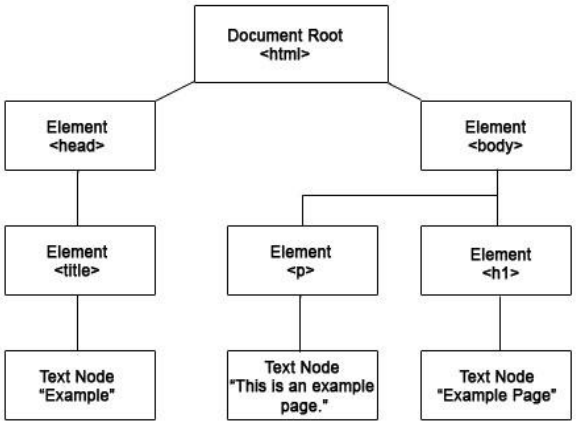

fig: DOM tree

For each element under the document root (<html>), there is an element node, and these element nodes have text nodes containing the text that is within the element. If there were an element with attributes, an attribute node would be created for that element and any text for the attribute would create a text node under that attribute node.

DOM methods:

x.getElementById(id):get the element with special id
x.getElementByTagName(name):get all elements with a specified tag name.
x.appendChild(node):insert a child node from x.
x.removeChild(node):remove a child node from x.
In the above list x is a node object.

Things to remember

- XSL stands forEXtensible Stylesheet Language.It describes how the XML document should be displayed.
- XSLT transforms an XML document into another XML document. XSLT uses XPath to navigate in XML documents.XSLT does this by transforming each XML element into an (X)HTML element.
- The element is used to build templates.The **match** attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document
- The element can be used to extract the value of an XML element and add it to the output stream of the transformation.
- The element allows you to do looping in XSLT. It can be used to select every XML element of a specified node-set
- To sort the output, simply add an element inside the element in the XSL file
- The element is used to put a conditional test against the content of the XML file. add the element inside the element in the XSL file
- The element is used in conjunction with and to express multiple conditional tests.
- XPath is a language and is used for finding the information in an XML document by using an addressing syntax based on a path through the document's logical structure or hierarchy
- XPath uses path expressions to select nodes in an XML document.

- XQuery is a functional,expression-oriented language which is designed to query XML data. It uses "FLWOR (FOR,LET,WHERE,ORDER BY,RETURN) expression" for performing Operation.
- SAX is an event-based sequential access parser API developed by the XML-DEV mailing list for XML documents.
- DOM is an application programming interface (API) for HTML and XML documents.The Document Object Model is a platform- independent and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

SAX vs. DOM

- In SAX, events are triggered when the XML is being **parsed**. When the parser is parsing the XML, and encounters a tag starting (e.g. <something>), then it triggers the tagStarted event (actual name of event might differ). Similarly when the end of the tag is met while parsing (</something>), it triggers tagEnded. Using a SAX parser implies you need to handle these events and make sense of the data returned with each event.
- In DOM, there are no events triggered while parsing. The entire XML is parsed and a DOM tree (of the nodes in the XML) is generated and returned. Once parsed, the user can navigate the tree to access the various data previously embedded in the various nodes in the XML.

Here in a more simple words:

DOM -Tree model parser (Object based) (Tree of nodes).

-DOM loads the file into the memory and then parse the file.
-Has memory constraints since it loads the whole XML file before parsing.
-DOM is read and write (can insert or delete the node).
-If the XML content is small then prefer DOM parser.
-Backward and forward search is possible for searching the tags and evaluation of the information inside the tags. So this gives the ease of navigation.
-Slower at run time.

SAX
-Event based parser (Sequence of events).
-SAX parses the file at it reads i.e. Parses node by node.
-No memory constraints as it does not store the XML content in the memory.
-SAX is read only i.e. can't insert or delete the node.
-Use SAX parser when memory content is large.
-SAX reads the XML file from top to bottom and backward navigation is not possible.
-Faster at run time.