XML (eXtensible Markup Language)

Introduction

XML stands for eXtensible Markup Language. It is designed to structure, store and transport data. XML tags are not predefined. We must define our own tags. XML is designed to be self-descriptive. XML and HTML were designed with different goals:

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

HTML is about displaying information, while XML is about storing and carrying information. It is just pure information wrapped in tags. Someone must write a piece of software to send, receive or display it.

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. A simple XML document is given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

In this document, 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 (note). The next 4 lines describe 4 **child elements** of the root (to, from, heading, and body). And finally the last line defines the end of the root element. You can assume, from this example, that the XML document contains a note to Tove from Jani. Hence, we can say that XML is pretty self-descriptive.

Elements and Attributes

An XML element is everything from (including) the element's start tag to (including) the element's end tag. An element can contain other elements, simple text or a mixture of both. Elements can also have attributes. Attributes provide additional information about elements. Attribute values must always be enclosed in quotes, but either single or double quotes can be used. In the example below, <bookstore> and <book> have **element contents**, because they contain other elements, <author> has **text content** because it contains text, and <book> has an **attribute** (category="CHILDREN").

```
<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>
```

XML documents form a tree structure that starts at "the root" and branches to "the leaves". 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. An element can have sub elements (**child elements**). 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).

XML Naming Rules

XML elements must follow these naming rules:

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

Best Naming Practices:

- Make names descriptive. Names with an underscore separator are nice: <first_name>, <last_name>.
- Names should be short and simple, like this: <book_title> not like this: <the_title_of_the_book>.
- Avoid "-" characters. If you name something "first-name," some software may think you want to subtract name from first.
- Avoid "." characters. If you name something "first.name," some software may think that "name" is a property of the object "first."
- Avoid ":" characters. Colons are reserved to be used for something called namespaces (more later).
- XML documents often have a corresponding database. A good practice is to use the naming rules of your database for the elements in the XML documents.
- Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software vendor doesn't support them.

Rules for Writing XML

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
- XML declaration is not a part of the XML document itself, and it has no closing tag
- XML Tags are Case Sensitive
- XML elements are defined using XML tags
- Opening and closing tags must be written with the same case
- XML Elements Must be Properly Nested

- XML Documents Must Have a Root Element, that is, XML documents must contain one element that is the **parent** of all other elements (the **root** element)
- XML Attribute Values Must be Quoted
- Use entity reference for some characters as shown below:

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

For example, <message>if salary &lt; 1000 then</message>
Note: Only the characters "<" and "&" are strictly illegal in XML. The greater than character is legal, but it is a good habit to replace it.

- The syntax for writing comments in XML is similar to that of HTML. For example, <!-- This is a comment -->
- White-space is Preserved in XML
- XML Stores New Line as LF (line feed)

XML Namespaces

We use namespaces 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. Name conflicts in XML can easily be avoided using a name prefix. For example, the XML below carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
  </h:tr>
</h:table>
<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

When using prefixes in XML, a **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*". For example,

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

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

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

```
<root
xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="http://www.w3schools.com/furniture">
<h:table>
  <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
  </h:tr>
</h:table>
<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
</root>
```

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

We can also define a default namespace for an element. In this case, we do not use prefixes in all the child elements. It has the following syntax: xmlns="*namespaceURI*". For example,

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

XML Schema

An XML Schema describes the structure of an XML document. The purpose of an XML Schema is to define the legal building blocks of an XML document. XML Schema is an XML-based alternative to DTD (Document Type Definition). The XML Schema language, also referred to as XML Schema Definition (XSD), is used to define XML schema. An XML Schema:

- defines elements that can appear in a document
- defines attributes that can appear in a document

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

XML Schemas are the successors of DTDs. We think that very soon 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

XML documents can have a reference to an XML Schema. For example consider the following “note.xml” file. This file has a reference the “note.xsd” schema.

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

The code fragment **xmlns="http://www.w3schools.com"** specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.w3schools.com" namespace.

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

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

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

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

```
            <xs:element name="from" type="xs:string"/>
            <xs:element name="heading" type="xs:string"/>
            <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

The <schema> element is the root element of every XML Schema. The <schema> element may contain some attributes. A schema declaration often looks something like this:

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

The code fragment **xmlns:xs="http://www.w3.org/2001/XMLSchema"** indicates that the elements and data types used in the schema come from the "http://www.w3.org/2001/XMLSchema" namespace. It also specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with **xs:**

The code fragment **targetNamespace="http://www.w3schools.com"** indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.w3schools.com" namespace.

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

The code fragment **elementFormDefault="qualified"** indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

Simple Types (Elements and Attributes)

XML Schemas define the elements of your XML files. A simple element is an XML element that contains only text. It cannot contain any other elements or attributes. The text can be of many different types. It can be one of the types included in the XML Schema definition or it can be a custom type that you can define yourself.

You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

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

- xs:string
- xs:decimal
- xs:integer
- xs:Boolean
- xs:date

- xs:time

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

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. A fixed value is also automatically assigned to the element, and you cannot specify another value. For example,
```
<xs:element name="color" type="xs:string" default="red"/>
<xs:element name="color" type="xs:string" fixed="red"/>
```

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

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

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

Restrictions on Values: The following example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 120:
```
<xs:element name="age">
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="120"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```
Restrictions on a Set of Values: To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint. The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:
```
<xs:element name="car">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

<u>Restrictions on a Series of Values:</u> To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint. The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

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

<u>Restrictions on Whitespace Characters:</u> 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>
```

<u>Restrictions on Length:</u> To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints. This example defines an element called "password" with a restriction. The value must be exactly eight characters:

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

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

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

Restrictions for Data types

Constraint	**Description**
Enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
Length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
Pattern	Defines the exact sequence of characters that are acceptable

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

Complex Types (Elements)

A complex element contains other elements and/or attributes. We can define a complex element in an XML Schema two different ways:

1. Element can be declared directly by naming the element. For example,

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

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

2. The element can have a type attribute that refers to the name of the complex type to use. For example,

```
<xs:element name="employee" type="personinfo"/>

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

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

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>

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

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

```
<xs:element name="employee" type="fullpersoninfo"/>
```

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

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

There are four kinds of complex elements: empty element, elements that contain only other elements, elements that contain only text, and elements that contain both other elements and text.

- **Empty elements:** An empty element cannot have contents, only attributes. For example, <product pid="1345"/>.
  You can define this empty element as follows:
  ```
  <xs:element name="product">
    <xs:complexType>
      <xs:attribute name="prodid" type="xs:positiveInteger"/>
    </xs:complexType>
  </xs:element>
  ```
  Or you can give the complexType element a name, and let the "product" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type as discussed before).
  ```
  <xs:element name="product" type="prodtype"/>
  <xs:complexType name="prodtype">
    <xs:attribute name="prodid" type="xs:positiveInteger"/>
  </xs:complexType>
  ```
- **Elements that contain only other elements:** This type contains an element that contains only other elements.For example,
  ```
  <person>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
  </person
  ```
  You can define the "person" element in a schema, like this:
  ```
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
  ```

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

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

```
<xs:element name="person" type="persontype"/>

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

- **Elements that contain only text:** This type can contain text and attributes: For example,
  `<shoesize country="france">35</shoesize>`
  This type contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simple content, you must define an extension OR a restriction within the simpleContent element.
  You can define the "shoesize" element in a schema, like this:

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

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

```
<xs:element name="shoesize" type="shoetype"/>

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

- **Elements that contain both other elements and text:** This type can contain attributes, elements, and text. For example,

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

The following schema declares the "letter" element:

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

Note: To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true".

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

```
<xs:element name="letter" type="lettertype"/>

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

Indicators: With indicators, we can control how elements are to be used in documents. There are seven indicators.

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

- **all:** This indicator specifies that the child elements can appear in any order, and that each child element must occur only once. For example,

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

- **choice:** This indicator specifies that either one child element or another can occur. For example,

```
<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:** This indicator specifies that the child elements must appear in a specific order. For example,

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

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

- **maxOccurs:** It specifies the maximum number of times an element can occur. For example,

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

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

- **minOccurs:** It specifies the minimum number of times an element can occur. For example,

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

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

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

3. Group indicators: These indicators are used to define related sets of elements.

- **group:** Element groups are defined with the group declaration. For example,

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

You must define an all, choice, or sequence element inside the group declaration. After you have defined a group, you can reference it in another definition, like this:

```
<xs:element name="person" type="personinfo"/>

<xs:complexType name="personinfo">
 <xs:sequence>
  <xs:group ref="persongroup"/>
  <xs:element name="country" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

- attributeGroup: We can define attribute groups with attributeGroup declaration. For example,

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

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

```
<xs:element name="person">
 <xs:complexType>
  <xs:attributeGroup ref="personattrgroup"/>
 </xs:complexType>
</xs:element>
```

<any> element: The <any> element enables us to extend the XML document with elements not specified by the schema. For example, the example below shows a declaration for the "person" element. By using the <any> element we can extend (after <lastname>) the content of "person" with any element defined in a separate schema.

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

<anyAttribute> element: This element enables us to extend the XML document with attributes not specified by the schema. For example, the example below shows a declaration for the "person" element. By using the <anyAttribute> element we can add any number of attributes defined in a separate schema to the "person" element.

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

DTD (Document Type Definition)

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

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

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

Internal DTD Declaration

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

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

!DOCTYPE note defines that the root element of this document is **note.**
!ELEMENT note defines that the **note** element contains four elements: "to,from,heading,body".
!ELEMENT to defines the **to** element to be of the type "#PCDATA".
!ELEMENT from defines the **from** element to be of the type "#PCDATA".
!ELEMENT heading defines the **heading** element to be of the type "#PCDATA".
!ELEMENT body defines the **body** element to be of the type "#PCDATA".

External DTD Declaration

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

Private external DTDs are identified by the keyword SYSTEM as shown before, and are intended for use by a single author or group of authors. Its syntax is: <!DOCTYPE root-element SYSTEM "DTD location">. For example,

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

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

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

Public external DTDs are identified by the keyword PUBLIC and are intended for broad use. Its syntax is: <!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">. The DTD_name follows the syntax: "prefix//owner_of_the_DTD//description_of_the_DTD//ISO 639_language_identifier". For example,

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

The following prefixes are allowed in the DTD name:

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

Defining Elements

Elements are the main building blocks of XML documents. In a DTD, elements are declared with an ELEMENT declaration with the following syntax.

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

Or

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

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

Elements with only parsed character data are declared with #PCDATA inside parentheses. Its syntax is: <!ELEMENT element-name (#PCDATA)>. For example, <!ELEMENT from (#PCDATA)>.
Elements with any content are declared with the category keyword ANY, can contain any combination of parsable data. Its syntax is: <!ELEMENT element-name ANY>. For example, <!ELEMENT note ANY>.

Elements with one or more children are declared with the name of the children elements inside parentheses. Its syntax is <!ELEMENT element-name (child1, child2,…)>. For example, <!ELEMENT note (to,from,heading,body)>.
When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children.

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

We can also declare **minimum one occurrence of an element**. Its syntax is <!ELEMENT element-name (child-name+)>. For example, <!ELEMENT note (message+)>. The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

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

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

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

Defining Attributes

In a DTD, attributes are declared with an **ATTLIST** declaration. An attribute declaration has the following syntax:
<!ATTLIST element-name attribute-name attribute-type default-value>
For example,
<!ATTLIST payment type CDATA "check">
And its XML example is
<payment type="check" />

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

Type	**Description**
CDATA	The value is character data (text that doesn’t contain markup)
(*en1*|*en2*|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names separated by whitespace
ENTITY	The name of an entity (which must be declared in the DTD)
ENTITIES	The value is a list of entities, separated by whitespace
NOTATION	The value is a name of a notation (which must be declared in the DTD)
xml:	The value is a predefined xml value

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

Value	**Explanation**
Value	The default value of the attribute. For example, <!ATTLIST square width CDATA "0">
#REQUIRED	The attribute is required. For example, <!ATTLIST person number CDATA #REQUIRED>

#IMPLIED	The attribute is not required (optional). For example, <!ATTLIST contact fax CDATA #IMPLIED>
#FIXED *value*	The attribute value is fixed. For example, <!ATTLIST sender company CDATA #FIXED "Microsoft">

XSLT

XSL stands for **EX**tensible **S**tylesheet **L**anguage, and is a style sheet language for XML documents. XSL describes how the XML document should be displayed. XSL consists of three parts: XSLT (a language for transforming XML documents), XPath (a language for navigating in XML documents), and XSL-FO(a language for formatting XML documents).

XSLT stands for **XSL Transformations**. XSLT is a language for transforming XML documents into XHTML documents or to other XML documents. 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 find information in an XML document. XPath is used to navigate through elements and attributes in XML documents. In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

Example:

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

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

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

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

XSL Style Sheet Declaration

The root element in the xsl document 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">
```

The XSL <xls:template> Element

The <xsl:template> element is used to build templates. An XSL style sheet consists of one or more set of rules that are called templates. A template contains rules to apply when a specified node is matched.

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, that is, 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 XSL <xls: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. For example, <xsl:value-of select="title"/>.

The XSL <xls:for-each> Element

The <xsl:for-each> element allows you to do looping in XSLT. The value of the **select** attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories. For example, <xsl:for-each select="catalog/cd">.

We can also **filter** the output from the XML file by adding a criterion to the select attribute in the <xsl:for-each> element. For example, <xsl:for-each select="catalog/cd[artist='Bob Dylan']">. Legal filter operators are:

- = (equal)
- != (not equal)
- &lt; less than
- &gt; greater than

The XSL <xls:sort> Element

The <xsl:sort> element is used to sort the output. To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file. For example,

```
<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>
```

The XSL <xls: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. For example,

```
<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>
```

The XSL <xls:choose> Element

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

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

```
      <td bgcolor="#ff00ff">
      <xsl:value-of select="artist"/></td>
    </xsl:when>
    <xsl: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, the **XML Path Language**, is a query language for selecting nodes from an XML document. It is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. Without XPath knowledge you will not be able to create XSLT documents.

XPath uses path expressions to navigate in XML documents. It also contains a library of standard functions for string values, numeric values, date and time comparison etc.

In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes. XML documents are treated as trees 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. The most useful path expressions are listed below:

Expression	**Description**
nodename	Selects all child nodes of the named node
/	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