**Schema**

A schema formally describes what a given XML document contains, in the same way a database schema describes the data that can be contained in a database (table structure, data types).

An XML schema describes the coarse (rough) shape of the XML document, what fields an element can contain, which sub-elements it can contain, and so forth. It also can describe the values that can be placed / allowed to be place into any element or attribute.

**An XML Schema describes the structure of an XML document.**

XML Schema language is also referred to as **XML Schema Definition (XSD)**.

**The purpose of an XML Schema is to define the legal building blocks of an XML document:**

* The elements and attributes that can appear in a document.
* The number of (and order of) child elements.
* Data types for elements and attributes.
* Default and fixed values for elements and attributes.

**A Note About Standards**undefined100%

**DTD** was the first formalized standard, but is rarely used anymore.

**XDR** was an early attempt by Microsoft to provide a more comprehensive standard than DTD. This standard has pretty much been abandoned now in favor of XSD.

**XSD** is currently the *de facto* standard for describing XML documents. There are two versions in use, 1.0 and 1.1, which are on the whole the same.

**Note:**

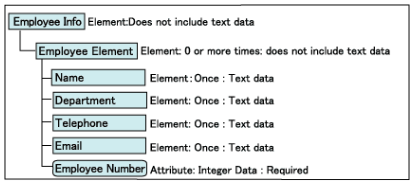
Traditionally, DTD has been the standard for XML schema definition; however, XML usage has expanded dramatically in core application systems, being tailored for a wide range of purposes for which DTD is not fully capable of supporting.

Given this development, the W3C recommended "**XML Schema**" as a schema definition language to replace DTD. The recommendation of XML Schema has spurred its adoption as a standard schema definition language.

\*spur - a thing that prompts or encourages someone; an incentive

**How to specify the Content to put into the XML document:**

1. The root element is "Employee\_Info"
2. As the content for "Employee\_Info," "Employee" occurs 0 or more times
3. As content of "Employee," "Name," "Department," "Telephone," and "Email" elements occur once in respective order
4. "Name," "Department," "Telephone," and "Email" content are text strings
5. "Employee" has an attribute called "Employee\_Number"
6. "Employee\_Number" content must be int type



**XSD**

**XML Schema Definition, commonly known as XSD**, is a way to describe precisely the XML language. XSD checks the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language.

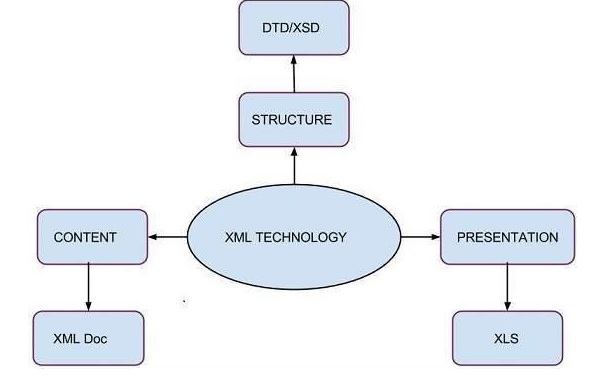
**An XML document can be defined as −**

**Well-formed** − If the XML document adheres to all the general XML rules such as tags must be properly nested, opening and closing tags must be balanced, and empty tags must end with '/>', then it is called as *well-formed*.

**OR**

**Valid** − An XML document said to be valid when it is not only *well-formed*, but it also conforms to available XSD that specifies which tags it uses, what attributes those tags can contain, and which tags can occur inside other tags, among other properties.

**The following diagram shows how XSD is used to structure XML documents −**



**Features :**

**Here is a list of some of the popular features of XSD −**

* XSDs can be extensible for future additions.
* XSD is richer and more powerful than DTD.
* XSD is written in XML.
* XSD supports data types.
* XSD supports namespaces.
* XSD is W3C recommendation.

## ****XML schema**** (XSD) Overview :

An XML schema, commonly known as an XML Schema Definition (XSD), formally describes what a given XML document can contain, in the same way that a database schema describes the data that can be contained in a database (i.e. table structure, data types, constraints etc.). The XML schema defines the shape, or structure, of an XML document, along with rules for data content and semantics such as what fields an element can contain, which sub elements it, can contain and how many items can be present. It can also describe the type and values that can be placed into each element or attribute. The XML data constraints are called **facets** and include rules such as min and max length.

### XML Schema Standards :

* **XML Schema Definition (XSD)** is currently the de facto standard for describing XML documents and is the XML Schema standard we will concentrate on in this tutorial. XSD is controlled by the World Wide Web Consortium (W3C). An XSD is itself an XML document, and there is even an XSD to describe the XSD standard.  
   \* **de facto** – in fact or in effect, whether by right or not.
* **Document Type Definition (DTD)** was the first formalized standard but has now, in most cases, been superseded by XSD.
* **XML Data Reduced (XDR)** was an early attempt but Microsoft to provide a more comprehensive standard than DTD. This standard has been phased out in the Microsoft products in favour of XSD.
* There are also a number of other schema standards such as **Schematron and RELAX NG.**

**Defining** **Elements**

Elements are the main building block of all XML documents, containing the data and determine the structure of the instance document.

**An element can be defined within an XSD as follows:**

***<xs:element name="x" type="y" />***

Each element definition within the XSD must have a **'name' property, which is the tag name** that will appear in the XML document. The **'type' property provides the description of what type of data can be contained** within the element when it appears in the XML document.

An element definition within the XSD must have a name property;

**Name property:**

It shows the name that will appear in the XML document.

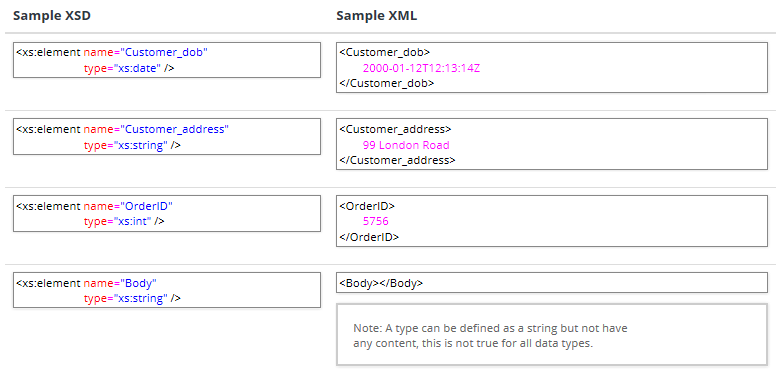
**Type Property:**

The type property provides the description of what can be contained within the element when it appears in the XML document. There are a number of predefined types, such as xs:string, xs:integer, xs:boolean or xs:. You also can create a user-defined type by using the <xs:simple type> and <xs:complexType> tags.

There are a number of predefined simple types, such as xs:string, xs:integer, xs:boolean and xs:date. Elements of these simple data types are said to have a **'simple content model'**, whereas elements that can contain other elements are said to have a '**complex content model**' and elements that can contain both have a **'mixed content model**'.

If we have set the type property for an element in the XSD, then the corresponding value in the XML document must be in the correct format for its given type otherwise this will cause a validation error when a validating parser attempts to parse the data from the XML document.

**Examples of simple elements and their XML data are shown below:**



The valid data values for the element in the XML document can be further constrained using the fixed and default properties.

**Default** means that if no value is specified in the XML document then the application reading the document, typically an XML parser or XML Data Binding Library, should use the default specified in the XSD.

**Fixed** means the value in the XML document can only have the value specified in the XSD.

For this reason it does not make sense to use both default and fixed in the same element definition, and is invalid to do so.

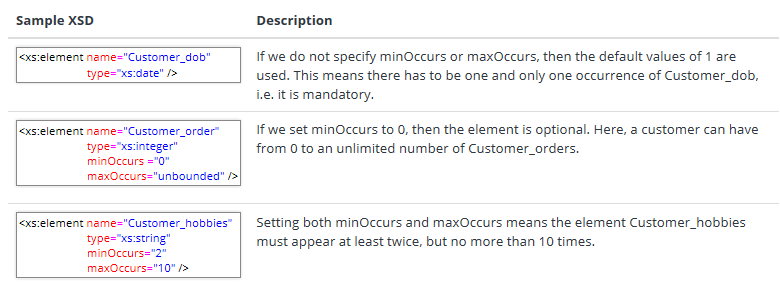
*<xs:element name="Customer\_name" type="xs:string" default="unknown" /><xs:element name="Customer\_location" type="xs:string" fixed=" UK" />*

## Specifying ****Cardinality****

Sometimes it is useful to add a constraint to allow a specific number of elements to appear at a specific point in an XML document, this is referred to as cardinality.

The cardinality is specified using the minOccurs and maxOccurs attributes, and allows an element to be specified as mandatory, optional, or can appear up to a set number of times. The default values for minOccurs and maxOccurs is 1. Therefore, if both the minOccurs and maxOccurs attributes are absent, as in all the previous examples, the element must appear once and once only.

'minOccurs' can be assigned any non-negative integer value (e.g. 0, 1, 2, 3... etc.), and 'maxOccurs' can be assigned any non-negative integer value or the special string constant "unbounded" meaning there is no maximum so the element can occur an unlimited number of times.



**Defining** **Complex Types**

A complex type is a container for other element definitions; this allows you to specify which child elements an element can contain. This allows you to provide some structure within your XML documents.

**Examples of this are:**

Here are some simple element definitions:

<xs:element name="Customer" type="xs:string" />

<xs:element name="Customer\_dob" type="xs:date" />

<xs:element name="Customer\_address" type="xs:string" />

<xs:element name="Supplier" type="xs:string" />

<xs:element name="Supplier\_phone" type="xs:integer" />

<xs:element name="Supplier\_address" type="xs:string" />

We can see that some of these elements should really be represented as child elements, "Customer\_dob" and "Customer\_address" belong to a parent element – "Customer". While "Supplier\_phone" and "Supplier\_address" belong to a parent element "Supplier". We can therefore re-write this in a more structured way:

*<xs:element name="Customer">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Dob" type="xs:date" />*

*<xs:element name="Address" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="Supplier">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Phone" type="xs:integer" />*

*<xs:element name="Address" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

#### What's changed?

* We created a definition for an element called "**Customer**".
* Inside the <xs:element> definition we added a <xs:complexType>. This is a container for other <xs:element> definitions, allowing us to build a simple hierarchy of elements in the resulting XML document.
* Note the contained elements for "Customer" and "Supplier" does not have a type specified as they do not extend or restrict an existing type; they are a new definition built from scratch.
* The <xs:complexType> element contains another new element <xs:sequence>, but more on these in a minute.
* The <xs:sequence> in turn contains the definitions for the two child elements "Dob" and "Address". Note the customer/supplier prefix has been removed as it is implied from its position within the parent element "Customer" or "Supplier".

So in plain English this is saying we can have an XML document that contains an element <Customer> which must have two child elements <Dob> and <Address>.

#### Example XML

*<Customer>*

*<Dob>2000-01-12T12:13:14Z</Dob>*

*<Address> 34 thingy street, someplace, sometown, ww1 8uu </Address>*

*</Customer>*

*<Supplier>*

*<Phone>0123987654</Phone>*

*<Address>22 whatever place, someplace, sometown, ss1 6gy </Address>*

*</Supplier>*

**Note:**

Complex Element is an XML element which can contain other elements and/or attributes.

**We can create a complex element in two ways −**

* Define a complex type and then create an element using the type attribute.
* Define a complex type directly by naming.

**Define a Complex Type and then create an element using type attribute:**

*<xs:complexType name = "StudentType">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

**Define a Complex Type directly by naming:**

*<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 = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

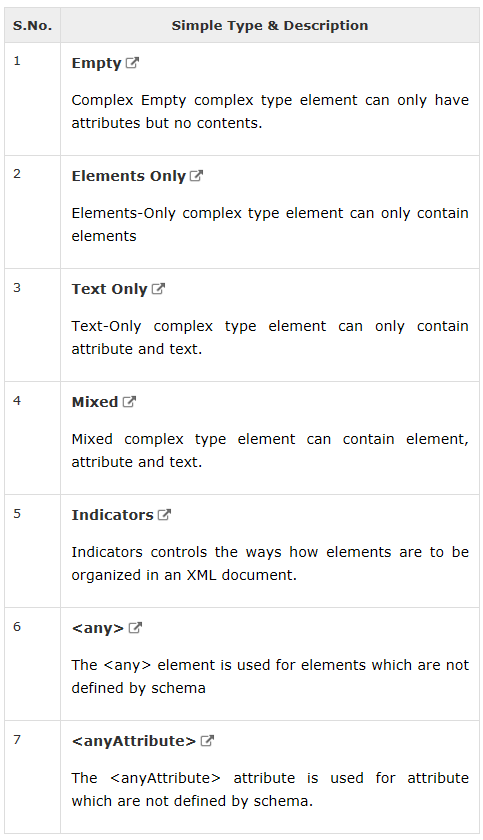
*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element>*

**Following is the list of Complex Types that XSD supports:**



## Complex Types Containing Elements Only

An XML element, "person”, which contains only other elements:

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

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

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

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

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

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

**XSD Text-Only Elements**

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

## Complex Text-Only Elements

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

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

**Tip:** Use the extension/restriction element to expand or to limit the base simple type for the element.

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

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

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

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

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

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

**XSD Mixed Content**

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

## Complex Types with Mixed Content

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

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

**The following schema declares the "letter" element:**

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

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

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

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

**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

## Defining ****Compositors****

There are three types of compositors <xs:sequence>, <xs:choice> and <xs:all>. These compositors allow us to determine how the child elements contained within them will appear within the XML document.

| **Compositor** | **Description** |
| --- | --- |
| **Sequence** | The child elements in the XML document MUST appear in the order they are declared in the XSD schema. |
| **Choice** | Only one of the child elements described in the XSD schema can appear in the XML document. |
| **All** | The child elements described in the XSD schema can appear in the XML document in any order. |

#### Notes

The compositors <xs:sequence> and <xs:choice> can be nested inside other compositors, and be given there own minOccurs and maxOccurs properties. This allows for quite complex combinations to be formed.

#### Example

The definitions of "Customer->Address" and "Supplier->Address" are currently not very usable as they are grouped into a single field. In the real world it would be better break this out into a few fields.

**Let's fix this by breaking it out using the same technique shown above:**

*<xs:element name="Customer">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Dob" type="xs:date" />*

*<xs:element name="Address">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="Supplier">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Phone" type="xs:integer" />*

*<xs:element name="Address">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

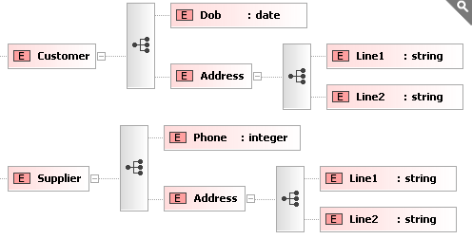
*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>  
</xs:element>*



This is much better, but we now have two definitions for address, which are the identical.

**Occurence Indicators**

* **maxOccurs** − Child element can occur only maxOccurs number of times.
* **minOccurs** − Child element must occur minOccurs number of times.

**Group Indicators**

* **Group** − Defines related set of elements.
* **attributeGroup** − Defines related set of attributes.

**Order Indicators**

Using <all> a student element can have firstname, lastname, nickname, and marks child element in any order in the XML document.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:all>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:all>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

Using <choice> a student element can have only one of firstname, lastname, nickname and marks child element in the XML document.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:choice>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:choice>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

Using <sequence>, a student element can have firstname, lastname, nickname and marks child element in the specified order only in the XML document.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

**Occurence Indicators**

Using <maxOccurs>, a student element can have maximum two nicknames in the XML document.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:all>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string" maxOccurs="2"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:all>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

Using <minOccurs>, a student element should have two nicknames in the XML document.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:all>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string" minOccurs="2"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:all>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

**Group Indicators**

The definition of "Customer->Address" and "Supplier->Address" are currently not very usable because they are grouped into a single field. In the real world, it would be better break this out into a few fields. You can fix this by breaking it out by using the same technique shown above:

*<xs:element name="Customer">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Dob" type="xs:date" />*

*<xs:element name="Address">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="Supplier">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Phone" type="xs:integer" />*

*<xs:element name="Address">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

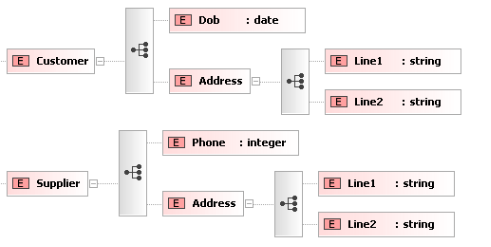
*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*



### Re-Use

It would make much more sense to have one definition of "Address" that could be used by both customer and supplier. You can do this by defining a complexType independently of an element, and giving it a unique name:

*<xs:complexType name="AddressType">*

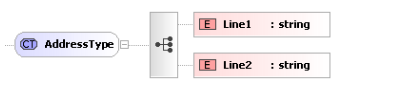
*<xs:sequence>*

*<xs:element name="Line1" type="xs:string"/>*

*<xs:element name="Line2" type="xs:string"/>*

*</xs:sequence>*

*</xs:complexType>*



You have now defined a <xs:complexType> that describes your representation of an address, so use it. Remember when you started looking at elements and I said you could define your own type instead of using one of the standard ones (xs:string, xs:integer)? Well, that's exactly what you are doing now.

*<xs:element name="Customer">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Dob" type="xs:date"/>*

*<xs:element name="Address" type="AddressType"/>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="supplier">*

*<xs:complexType>*

*<xs:sequence>*

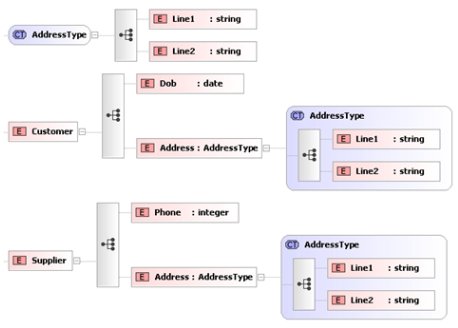
*<xs:element name="address" type="AddressType"/>*

*<xs:element name="phone" type="xs:integer"/>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*



The advantage should be obvious. Instead of having to define Address twice (once for Customer and once for Supplier), you have a single definition. This makes maintenance simpler ie if you decide to add "Line3" or "Postcode" elements to your address; you only have to add them in one place.

### Example XML

*<Customer>*

*<Dob> 2000-01-12T12:13:14Z </Dob>*

*<Address>*

*<Line1>34 thingy street, someplace</Line1>*

*<Line2>sometown, w1w8uu </Line2>*

*</Address>*

*</Customer>*

*<Supplier>*

*<Phone>0123987654</Phone>*

*<Address>*

*<Line1>22 whatever place, someplace</Line1>*

*<Line2>sometown, ss1 6gy </Line2>*

*</Address>*

*</Supplier>*

**Note:** Only complex types defined globally (because children of the <xs:schema> element can have their own name and be re-used throughout the schema). If they are defined inline within an <xs:element>, they can not have a name (anonymous) and can not be re-used elsewhere.

<group> is used to group a related set of elements. Here we've created a group of part of name and then used this group to define a **student** element.

*<xs:group name = "infogroup">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "birthdate" type = "xs:date"/>*

*</xs:sequence>*

*</xs:group>*

*<xs:element name = "student" type = "studentType"/>*

*<xs:complexType name = "studentType">*

*<xs:sequence>*

*<xs:group ref = "infogroup"/>*

*<xs:element name = "marks" type = "xs:integer"/>*

*</xs:sequence>*

*</xs:complexType>*

<attributeGroup> is used to group a related set of attribute. Here we've created a group of part of name and then used this group to define attributes for **student** element.

*<xs:attributeGroup name = "infogroup">*

*<xs:sequence>*

*<xs:attribute name = "firstname" type = "xs:string"/>*

*<xs:attribute name = "lastname" type = "xs:string"/>*

*<xs:attribute name = "birthdate" type = "xs:date"/>*

*</xs:sequence>*

*</xs:attributeGroup>*

*<xs:element name = "student" type = "studentType"/>*

*<xs:complexType name = "studentType">*

*<xs:sequence>*

*<xs:attributeGroup ref = "infogroup"/>*

*<xs:element name = "marks" type = "xs:integer"/>*

*</xs:sequence>*

*</xs:complexType>*

## Defining ****Global Types****

It would make much more sense to have a single definition for "Address", which could then be used by both customer and supplier. We can do this by defining a complexType independently of an element, and giving it a unique name:

*<xs:complexType name="AddressType">*

*<xs:sequence>*

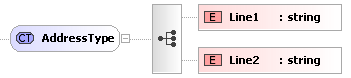
*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

**The previous XSD definitions are shown graphically in Liquid XML Studio as follows:**

[](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/global-complex-type.png)

We have now defined a <xs:complexType> that describes our representation of an address, so let's use it. Earlier, when we started looking at elements, we said you could define your own types instead of using one of the standard types such as xs:string or xs:integer, and that is exactly what were now doing.

*<xs:element name="Customer">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Dob" type="xs:date" />*

*<xs:element name="Address" type="AddressType" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="Supplier">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Phone" type="xs:integer" />*

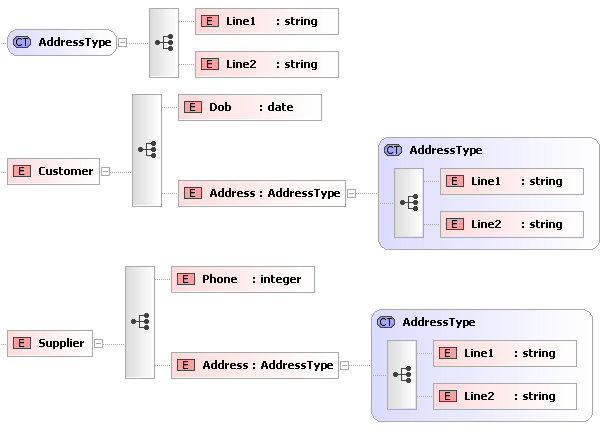
*<xs:element name="Address" type="AddressType" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

**The previous XSD definitions are shown graphically in Liquid XML Studio as follows:**

[](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/referencing-global-types.png)

Hopefully, the advantages are obvious. Instead of having to define Address twice (once for Customer and once for Supplier) we now have a single definition. This makes maintenance simpler, i.e. if you decide to add "Line3" or "Postcode" elements to your address you only have to add them in one place.

#### Example XML

*<Customer>*

*<Dob>2000-01-12T12:13:14Z</Dob>*

*<Address>*

*<Line1>34 thingy street, someplace</Line1>*

*<Line2>sometown, ww1 8uu</Line2>*

*</Address>*

*</Customer>*

*<Supplier>*

*<Phone>0123987654</Phone>*

*<Address>*

*<Line1>22 whatever place, someplace</Line1>*

*<Line2>sometown, ss1 6gy</Line2>*

*</Address>*

*</Supplier>*

#### Notes

Only complex types defined globally (as children of the <xs:schema> element can have their own name and be re-used throughout the schema). If they are defined inline within an <xs:element> they can not have a name (anonymous) and can not be reused elsewhere.

## Defining ****Attributes****

An attribute provides extra information within an element. Attributes have name and type properties and are defined within an XSD as follows:

***<xs:attribute name="x" type="y" />***

An Attribute can appear 0 or 1 times within a given element in the XML document. Attributes are either optional or mandatory (by default they are optional). The "use" property in the XSD definition is used to specify if the attribute is optional or mandatory.

**So the following are equivalent:**

***<xs:attribute name="ID" type="xs:string" />***

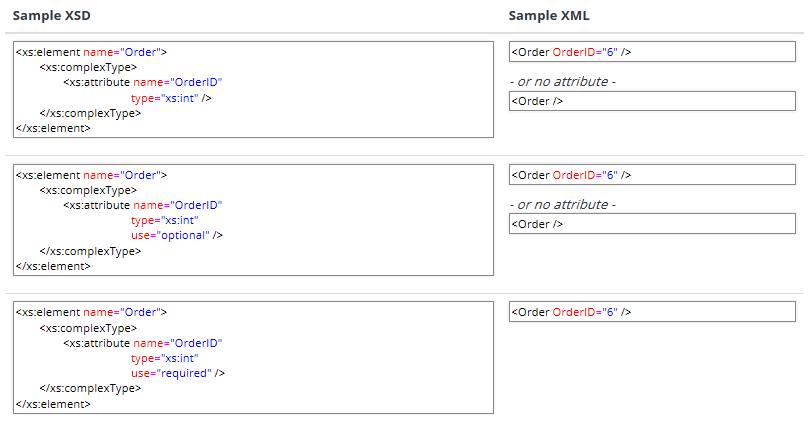
***<xs:attribute name="ID" type="xs:string" use="optional" />***

**The previous XSD definitions are shown graphically in Liquid XML Studio as follows:**

[Defining Attributes](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/defining-attributes.png)

To specify that an attribute must be present, use = "required" (Note: use may also be set to "prohibited", but we'll come to that later).

An attribute is typically specified within the XSD definition for an element, nesting the attribute in the element. Attributes can also be specified globally and then referenced (but more about this later).



The default and fixed attributes can be specified within the XSD attribute specification (in the same way as they are for elements).

## Using ****Mixed Content****

So far we have seen how an element can contain data, other elements and attributes. Elements can also contain a combination of all of these. You can also mix elements and data. You can specify this in the XSD schema by setting the mixed property.

*<xs:element name="MarkedUpDesc">*

*<xs:complexType mixed="true">*

*<xs:sequence>*

*<xs:element name="Bold" type="xs:string" />*

*<xs:element name="Italic" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

**A sample XML document could look like this:**

*<MarkedUpDesc>*

*This is an <Bold>Example</Bold> of <Italic>Mixed</Italic> Content, Note there are elements mixed in with the elements data.*

*</MarkedUpDesc>*

## Should I use an ****Element**** or an ****Attribute****?

It is often confusing when to use an element as opposed to using an attribute within your XML Schema. Some designers have the opinion that elements describe data whereas attributes describe the Meta data, others would say that attributes are used for small pieces of data such as an order id, but really it is personal taste with no hard and fast rules as to when to use an attribute.

A good rule of thumb might be to only use an attribute if it can be considered an aggregate of the parent element that relies on the parent to make sense. Whereas a child Element may be perfectly happy to exist outside of the parent element, in other words it is a composite item that has a relationship with the parent element.

So an element named Shape may have an attribute named Colour, i.e. Meta data about the Shape, and a child element that represents a sequence of elements named Point, an independent structure of data.

**Sample XML Schema (XSD):**

*<xs:element name="Shape">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="Point" minOccurs="1" maxOccurs="unbounded">*

*<xs:complexType>*

*<xs:attribute name="x" type="xs:int" />*

*<xs:attribute name="y" type="xs:int" />*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*<xs:attribute name="Colour" type="xs:string" />*

*</xs:complexType>*

*</xs:element>*

**Sample XML:**

*<Shape Colour="Black">*

*<Point x="0" y="0" />*

*<Point x="100" y="0" />*

*<Point x="50" y="50" />*

*</Shape>*

Using attributes as containers for data will mean you end up creating documents that are difficult to read and maintain, so try to use elements to describe your data.

**Some limitations and possible problems with using attributes include:**

* Unlike elements, attributes cannot contain multiple values.
* Attributes are not easily expandable to incorporate future changes to the schema.
* Attributes cannot describe structure whereas child elements can contain a whole variety of child structures.

## Should I use ****Mixed Content****?

Mixed content is something you should try to avoid when creating your XML schema. It is used extensively on the web as part of the xHTML standard where is makes sense as the tags are marking up the content. However, it is difficult to parse and it can lead to unforeseen complexity in the XML document's data.

## ****Best Practices**** when Writing XML Schema (XSD)

* All Element and attributes should use Upper Camel Case (UCC), e.g. (PostalAddress), and should avoid hyphens, spaces or other syntax.
* Readability is more important than tag length up to a point. There is always a line to draw between document size and readability, wherever possible favour readability.
* Avoid abbreviations and acronyms for element, attribute, and type names. Exceptions should be well known within your business area e.g. ID (Identifier), and POS (Point of Sale).
* Postfix all types with the name 'Type', e.g. AddressType. Several standards include Elements and ComplexTypes with the same name which leads to confusion.
* Enumerations should use names not numbers and the values should again be UCC.
* Names should not include the name of the containing structure, e.g. CustomerName should be Name within the parent element Customer.
* Only produce complexTypes or simpleTypes for types that are likely to be re-used. If the structure will only exists in one place define it inline with an anonymous [complexType](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes#complex-types).
* Avoid the use of [mixed content](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-conventions#mixed-content).
* Only define root level elements if the element is capable of being the root element in an XML document. If you want the element to have Global scope, create a root level ComplexType or SimpleType instead.
* Use consistent namespace aliases and **avoid using the standard defined prefix**:
  + **xml** (defined in XML standard)
  + **xmlns** (defined in Namespaces in XML standard)
  + **xs** (defined as http://www.w3.org/2001/XMLSchema)
  + **xsi** (defined as http://www.w3.org/2001/XMLSchema-instance)
* Try to think about versioning early on in your schema design. If its important for a new versions of a schema to be backwardly compatible, then all additions to the schema should be optional. If it is important that existing products should be able to read newer versions of a given document, then consider adding any and anyAttribute entries to the end of your definitions.
* Define a targetNamespace in your schema, this better identifies your schema and can make things easier to modularize and re-use.
* Set elementFormDefault="qualified" in the schema element of your schema. This makes qualifying the name spaces in the resulting XML simpler to read.
* Using an [XML Schema Editor](https://www.liquid-technologies.com/xml-schema-editor) will help you to produce valid XML schema saving you a lot of time.

**Example:**

***XSD:***

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

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

*<xs:element name="Shape">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="UAKAddress" type="UAKAddressType"/>*

*<xs:element name="USAAddress" type="USAAddressType"/>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:complexType name="UAKAddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string"/>*

*<xs:element name="Line2" type="xs:string"/>*

*<xs:element name="Country" type="xs:string"/>*

*<xs:element name="Pincode" type="xs:string"/>*

*</xs:sequence>*

*</xs:complexType>*

*<xs:complexType name="USAAddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string"/>*

*<xs:element name="Line2" type="xs:string"/>*

*<xs:element name="State" type="xs:string"/>*

*<xs:element name="Zipcode" type="xs:string"/>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:schema>*

**XML:**

*<Shape xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="H:\Test\Test1.xsd">*

*<UAKAddress>*

*<Line1>34 thingy street</Line1>*

*<Line2>someplace</Line2>*

*<Country>somerset</Country>*

*<Pincode>w1w8uu</Pincode>*

*</UAKAddress>*

*<USAAddress>*

*<Line1>234 Lancaster Av</Line1>*

*<Line2>Smallville</Line2>*

*<State>Florida</State>*

*<Zipcode>34543</Zipcode>*

*</USAAddress>*

*</Shape>*

**Data Types** **Overview**

It is often useful to be able to take the definition for an existing entity, and extend it to add more specific information. In most modern development languages, such as C++, C# or Java, we would call this specialization, inheritance or sub classing.

This same concept also exists in the XML Schema standard, allowing us to take an existing type definition and extend it. Types defined in an XSD can also be restricted (although this behaviour has no real parallel in most development languages).

**General Data Types**

|  |  |
| --- | --- |
| Name | Explanation |
| xs:integer | Integers (infinite precision) |
| xs:positiveInteger | Positive integers (infinite precision) |
| xs:negativeInteger | Negative integers (infinite precision) |
| xs:nonPositiveInteger | Negative integers including 0 (infinite precision) |
| xs:nonNegativeInteger | Positive integers including 0 (infinite precision) |
| xs:byte | Integer represented by 8 bits |
| xs:unsignedByte | Integer represented by 8 bits (no symbols) |
| xs:short | Integer represented by 16 bits |
| xs:unsignedShort | Integer represented by 16 bits (no symbols) |
| xs:int | Integer represented by 32 bits |
| xs:unsignedInt | Integer represented by 32 bits (no symbols) |
| xs:long | Integer represented by 64 bits |
| xs:unsignedLong | Integer represented by 64 bits (no symbols) |
| xs:decimal | Decimal number (infinite precision) |
| xs:float | Single-precision floating-point number (32-bit) |
| xs:double | Double-precision floating-point number (64-bit) |
| xs:Boolean | Boolean value |
| xs:string | Arbitrary text string |

**Types Representing Dates and Times**

|  |  |
| --- | --- |
| Name | Explanation |
| xs:time | Time of day |
| xs:dateTime | Date and time of day |
| xs:date | Date |
| xs:gYear | Year |
| xs:gYearMonth | Year and month |
| xs:gMonth | Month |
| xs:gMonthDay | Month and day |
| xs:gDay | Day |

**DTD-Compatible Types**

|  |  |
| --- | --- |
| Name | Explanation |
| xs:ID | XML 1.0 Specification ID type |
| xs:IDREF | XML 1.0 Specification IDREF type |
| xs:IDREFS | XML 1.0 Specification IDREFS type |
| xs:ENTITY | XML 1.0 Specification ENTITY type |
| xs:ENTITIES | XML 1.0 Specification ENTITIES type |
| xs:NOTATION | XML 1.0 Specification NOTATION type |
| xs:NMTOKEN | XML 1.0 Specification NMTOKEN type |
| xs:NMTOKENS | XML 1.0 Specification NMTOKENS type |

**XSD – String**

String data types are used to represent characters in the XML documents.

**<xs:string> data type**

The <xs:string> data type can take characters, line feeds, carriage returns, and tab characters. The XML processor does not replace line feeds, carriage returns, and tab characters in the content with space and keep them intact. For example, multiple spaces or tabs are preserved during display.

**<xs:string> Example**

Element declaration in xsd −

*<xs:element name = "name" type = "xs:string"/>*

**Element usage in xml −**

*<name>Dinkar</name>*

*<name>Dinkar Kad</name>*

**<xs:token> data type**

The <xs:token> data type is derived from <string> data type and can take characters, line feeds, carriage returns, and tab characters. XML processor removes line feeds, carriage returns, and tab characters in the content and keep them intact. For example, multiple spaces or tabs are removed during display.

**<xs:token> Example**

Element declaration in xsd −

*<xs:element name = "name" type = "xs:token"/>*

**Element usage in xml −**

*<name>Dinkar</name>*

*<name>Dinkar Kad</name>*

**String Data Types**

Following is the list of commonly used data types which are derived from <string> data type.

|  |  |
| --- | --- |
| **S.No.** | **Name & Description** |
| 1 | **ID**  Represents the ID attribute in XML and is used in schema attributes. |
| 2 | **IDREF**  Represents the IDREF attribute in XML and is used in schema attributes. |
| 3 | **language**  Represents a valid language id |
| 4 | **Name**  Represents a valid XML name |
| 5 | **NMTOKEN**  Represents a NMTOKEN attribute in XML and is used in schema attributes. |
| 6 | **normalizedString**  Represents a string that does not contain line feeds, carriage returns, or tabs. |
| 7 | **string**  Represents a string that can contain line feeds, carriage returns, or tabs. |
| 8 | **token**  Represents a string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces |

**Restrictions**

Following types of restrictions can be used with String data types −

* enumeration
* length
* maxLength
* minLength
* pattern
* whiteSpace

**XSD - Date Time**

Date and Time data types are used to represent date and time in the XML documents.

**<xs:date> data type**

The <xs:date> data type is used to represent date in YYYY-MM-DD format.

* ***YYYY*** *− represents year*
* ***MM*** *− represents month*
* ***DD*** *− represents day*

**<xs:date> Example**

Element declaration in XSD −

*<xs:element name = "birthdate" type = "xs:date"/>*

Element usage in XML −

*<birthdate>1980-03-23</birthdate>*

**<xs:time> data type**

The <xs:time> data type is used to represent time in hh:mm:ss format.

* **hh** − represents hours
* **mm** − represents minutes
* **ss** − represents seconds

**<xs:time> Example**

Element declaration in XSD −

*<xs:element name = "startTime" type = "xs:time"/>*

Element usage in XML −

*<startTime>10:20:15</startTime>*

**<xs:datetime> data type**

The <xs:datetime> data type is used to represent date and time in YYYY-MM-DDThh:mm:ss format.

* ***YYYY*** *− represents year*
* ***MM*** *− represents month*
* ***DD*** *− represents day*
* ***T*** *− represents start of time section*
* ***hh*** *− represents hours*
* ***mm*** *− represents minutes*
* ***ss*** *− represents seconds*

**<xs:datetime> Example**

Element declaration in XSD −

*<xs:element name = "startTime" type = "xs:datetime"/>*

Element usage in XML −

*<startTime>1980-03-23T10:20:15</startTime>*

**<xs:duration> data type**

The <xs:duration> data type is used to represent time interval in PnYnMnDTnHnMnS format. Each component is optional except P.

* ***P*** *− represents year*
* ***nY*** *− represents month*
* ***nM*** *− represents day*
* ***nD*** *− represents day*
* ***T*** *− represents start of time section*
* ***nH*** *− represents hours*
* ***nM*** *− represents minutes*
* ***nS*** *− represents seconds*

**<xs:duration> Example**

Element declaration in XSD −

*<xs:element name = "period" type = "xs:duration"/>*

Element usage in xml to represent period of 6 years, 3 months, 10 days and 15 hours.

*<period>P6Y3M10DT15H</period>*

**Date Data Types**

Following is the list of commonly used date data types.

|  |  |
| --- | --- |
| **S.No.** | **Name & Description** |
| 1. | **date**  Represents a date value |
| 2. | **dateTime**  Represents a date and time value |
| 3. | **duration**  Represents a time interval |
| 4. | **gDay**  Represents a part of a date as the day (DD) |
| 5. | **gMonth**  Represents a part of a date as the month (MM) |
| 6. | **gMonthDay**  Represents a part of a date as the month and day (MM-DD) |
| 7. | **gYear**  Represents a part of a date as the year (YYYY) |
| 8. | **gYearMonth**  Represents a part of a date as the year and month (YYYY-MM) |
| 9. | **time**  Represents a time value |

**Restrictions**

Following types of restrictions can be used with Date data types −

* enumeration
* maxExclusive
* maxInclusive
* minExclusive
* minInclusive
* pattern
* whiteSpace

**XSD - Numeric Data Types**

Numeric data types are used to represent numbers in XML documents.

**<xs:decimal> data type**

The <xs:decimal> data type is used to represent numeric values. It supports decimal numbers up to 18 digits.

**<xs:decimal> Example**

**Element declaration in XSD −**

*<xs:element name = "score" type = "xs:decimal"/>*

**Element usage in XML −**

*<score>9.12</score>*

**<xs:integer> data type**

The <xs:integer> data type is used to represent integer values.

**<xs:integer> Example**

Element declaration in XSD −

*<xs:element name = "score" type = "xs:integer"/>*

**Element usage in XML −**

*<score>9</score>*

**Numeric Data Types**

Following is the list of commonly used numeric data types.

|  |  |
| --- | --- |
| **S.No.** | **Name & Description** |
| 1. | **byte**  A signed 8 bit integer |
| 2. | **decimal**  A decimal value |
| 3. | **int**  A signed 32 bit integer |
| 4. | **integer**  An integer value |
| 5. | **long**  A signed 64 bit integer |
| 6. | **negativeInteger**  An integer having only negative values (..,-2,-1) |
| 7. | **nonNegativeInteger**  An integer having only non-negative values (0,1,2,..) |
| 8. | **nonPositiveInteger**  An integer having only non-positive values (..,-2,-1,0) |
| 9. | **positiveInteger**  An integer having only positive values (1,2,..) |
| 10. | **short**  A signed 16 bit integer |
| 11. | **unsignedLong**  An unsigned 64 bit integer |
| 12. | **unsignedInt**  An unsigned 32 bit integer |
| 13. | **unsignedShort**  An unsigned 16 bit integer |
| 14. | **unsignedByte**  An unsigned 8 bit integer |

**Restrictions**

Following types of restrictions can be used with Date data types −

* *enumeration*
* *fractionDigits*
* *maxExclusive*
* *maxInclusive*
* *minExclusive*
* *minInclusive*
* *pattern*
* *totalDigits*
* *whiteSpace*

**XSD - Miscellaneous Data Types**

XSD has a few other important data types, such as **Boolean, binary,** and **anyURI.**

**<xs:boolean> data type**

The <xs:boolean> data type is used to represent true, false, 1 (for true) or 0 (for false) value.

**<xs:boolean> Example**

Element declaration in XSD −

*<xs:element name = "pass" type = "xs:boolean"/>*

**Element usage in XML −**

*<pass>false</pass>*

**Binary data types**

The Binary data types are used to represent binary values. Two binary types are common in use.

* **base64Binary** − represents base64 encoded binary data
* **hexBinary** − represents hexadecimal encoded binary data

**<xs:hexbinary> Example**

**Element declaration in XSD −**

*<xs:element name = "blob" type = "xs:hexBinary"/>*

**Element usage in XML −**

*<blob>9FEEF</blob>*

**<xs:anyURI> data type**

The <xs:anyURI> data type is used to represent URI.

**<xs:anyURI> Example**

**Element declaration in XSD −**

*<xs:attribute name = "resource" type = "xs:anyURI"/>*

**Element usage in XML −**

*<image resource = "http://www.tutorialspoint.com/images/smiley.jpg" />*

**Numeric Data Types**

Following is the list of commonly used numeric data types.

|  |  |
| --- | --- |
| **S.No.** | **Name & Description** |
| 1. | **byte**  A signed 8 bit integer |
| 2. | **decimal**  A decimal value |
| 3. | **int**  A signed 32 bit integer |
| 4. | **integer**  An integer value |
| 5. | **long**  A signed 64 bit integer |
| 6. | **negativeInteger**  An integer having only negative values (..,-2,-1) |
| 7. | **nonNegativeInteger**  An integer having only non-negative values (0,1,2,..) |
| 8. | **nonPositiveInteger**  An integer having only non-positive values (..,-2,-1,0) |
| 9. | **positiveInteger**  An integer having only positive values (1,2,..) |
| 10. | **short**  A signed 16 bit integer |
| 11. | **unsignedLong**  An unsigned 64 bit integer |
| 12. | **unsignedInt**  An unsigned 32 bit integer |
| 13. | **unsignedShort**  An unsigned 16 bit integer |
| 14. | **unsignedByte**  An unsigned 8 bit integer |

**Restrictions**

Following types of restrictions can be used with Miscellaneous data types except on boolean data type −

* *enumeration*
* *length*
* *maxLength*
* *minLength*
* *pattern*
* *whiteSpace*

|  |  |  |
| --- | --- | --- |
| Simple Type | Examples (delimited by commas) | Explanation |
| string | Confirm this is electric | A text string |
| base64Binary | GpM7 | Base86 encoded binary data |
| hexBinary | 0FB7 | HEX encoded binary data |
| integer | ...-1, 0, 1, ... |  |
| positiveInteger | 1, 2, ... |  |
| negativeInteger | ... -2, -1 |  |
| nonNegativeInteger | 0, 1, 2, ... |  |
| long | -9223372036854775808, ... -1, 0, 1, ... 9223372036854775807 |  |
| decimal | -1.23, 0, 123.4, 1000.00 |  |
| float | -INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN |  |
| boolean | true, false, 1, 0 |  |
| duration | P1Y2M3DT10H30M12.3S | 1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds |
| dataTime | 1999-05-31T13:20:00.000-05:00 | May 31st 1999 at 1.20pm Eastern Standard Time |
| date | 1999-05-31 |  |
| time | 13:20:00.000, 13:20:00.000-05:00 |  |
| gYear | 1999 |  |
| Name | shipTo | XML 1.0 Name type |
| QName | po:USAddress | XML Namespace QName |
| anyURI | <http://www.example.com/> |  |
| language | en-GB, en-US, fr | valid values for xml:lang as defined in XML 1.0 |

## ****Extending**** Complex Types

It is possible to take an existing <xs:complexType> and extend it. Let's see how this may be useful with an example.

Looking at the AddressType that we defined earlier, let's assume our company has now gone international and we need to capture country specific addresses. In this case we need specific information for UK addresses (County and Postcode), and for US addresses (State and ZipCode).

**So we can take our existing definition of address and extend it as follows:**

*<xs:complexType name="AddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*<xs:complexType name="UKAddressType">*

*<xs:complexContent>*

*<xs:extension base="AddressType">*

*<xs:sequence>*

*<xs:element name="County" type="xs:string" />*

*<xs:element name="Postcode" type="xs:string" />*

*</xs:sequence>*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

*<xs:complexType name="USAddressType">*

*<xs:complexContent>*

*<xs:extension base="AddressType">*

*<xs:sequence>*

*<xs:element name="State" type="xs:string" />*

*<xs:element name="Zipcode" type="xs:string" />*

*</xs:sequence>*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

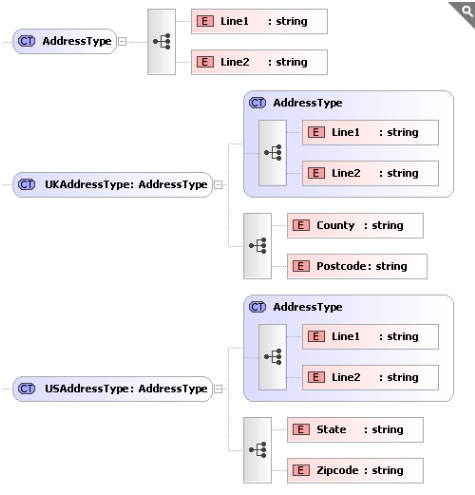
Notice each of the two new address types extend the original 'base' address type using:

*<xs:extension base="AddressType">*

The newly introduced construct <xs:extension> indicates that we are extending an existing type, and specifies the type itself. There is also another new construct, the <xs:complexContent> element, which is just a container for the extension.

So to reiterate, we are defining a new <xs:complexType> called "USAddressType", this extends the existing type "AddressType", and adds to it a sequence containing the elements "State", and "Zipcode".

**This is clearer when viewed graphically:**



**We can now use these new types as follows:**

*<xs:element name="UKAddress" type="UKAddressType" />*

*<xs:element name="USAddress" type="USAddressType" />*

**Sample XML for these elements may look like this:**

*<UKAddress>*

*<Line1>34 thingy street</Line1>*

*<Line2>someplace</Line2>*

*<County>somerset/County>*

*<Postcode>w1w8uu</Postcode>*

*</UKAddress>*

*<USAddress>*

*<Line1>234 Lancaster Av</Line1>*

*<Line2>Smallville</Line2>*

*<State>Florida</State>*

*<Zipcode>34543</Zipcode>*

*</USAddress>*

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

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

*<xs:complexType name="CommonType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string"/>*

*<xs:element name="Line2" type="xs:string"/>*

*</xs:sequence>*

*</xs:complexType>*

*<xs:element name="Shape">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="UAKAddress" type="UAKAddressType"/>*

*<xs:element name="USAAddress" type="USAAddressType"/>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:complexType name="UAKAddressType">*

*<xs:complexContent>*

*<xs:extension base="CommonType">*

*<xs:sequence>*

*<xs:element name="Country" type="xs:string"/>*

*<xs:element name="Pincode" type="xs:string"/>*

*</xs:sequence>*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

*<xs:complexType name="USAAddressType">*

*<xs:complexContent>*

*<xs:extension base="CommonType">*

*<xs:sequence>*

*<xs:element name="State" type="xs:string"/>*

*<xs:element name="Zipcode" type="xs:string"/>*

*</xs:sequence>*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

*</xs:schema>*

*<Shape xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="H:\Test\Test1.xsd">*

*<UAKAddress>*

*<Line1>34 thingy street</Line1>*

*<Line2>someplace</Line2>*

*<Country>somerset</Country>*

*<Pincode>w1w8uu</Pincode>*

*</UAKAddress>*

*<USAAddress>*

*<Line1>234 Lancaster Av</Line1>*

*<Line2>Smallville</Line2>*

*<State>Florida</State>*

*<Zipcode>34543</Zipcode>*

*</USAAddress>*

*</Shape>*

## ****Restricting**** Complex Types

The previous section showed how to take an existing <xs:complexType> definition, and extend it to create new types. But there is another option here, instead of adding to the type, we could restrict it.

Taking the same AddressType example, we can create a new type called "InternalAddressType". Let's assume "InternalAddressType" only needs Address->Line1.

*<xs:complexType name="AddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" minOccurs="0" />*

*</xs:sequence>*

*</xs:complexType>*

*<xs:complexType name="InternalAddressType">*

*<xs:complexContent>*

*<xs:restriction base="AddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*</xs:sequence>*

*</xs:restriction>*

*</xs:complexContent>*

*</xs:complexType>*

**Notice the new address type restricts the original 'base' address type using:**

*<xs:restriction base="AddressType">*

We are defining a new type "InternalAddressType". The <xs:restriction> element says we are restricting the existing type "AddressType" , and we are only allowing the existing child element "Line1" to be used in this new definition. The <xs:complexContent> element is just a container for the restriction.

We also need to make a small modification to the base type as Derivation by restriction does not allow you to add or omit elements (unless they are optional in the base type), it simply allows you to restrict their valid values e.g. set a default value or set type="string" where previously no type was specified. So we must change "Line2" to have minOccurs="0".

Note: As we are restricting an existing type the only definitions that can appear in the <xs:restriction> are a sub set of the ones defined in the base type "AddressType". They must also be enclosed in the same compositor (in this case a sequence) and appear in the same order.

**We can now use this new type as follows:**

*<xs:element name="InternalAddress" type="InternalAddressType" />*

**Sample XML for this element may look like this:**

*<InternalAddressType>*

*<Line1>Desk 4, Second Floor/<Line1>*

*</InternalAddressType>*

## Using the ****xsi:type**** Attribute

We have just shown how we can create new types based on existing one. This in itself is pretty useful, and will potentially reduce the amount of complexity in your schemas, making them easier to maintain and understand. However there is an aspect to this that has not yet been covered. In the above examples we created 3 new types (UKAddressType, USAddressType and InternalAddressType), all based on AddressType.

So, if we have an element that explicitly specifies it is of type "UKAddressType", then "UKAddressType" is what must appear in the XML document.

But if an element specifies its of type "AddressType", then any of the 4 types can appear in the XML document (UKAddressType, USAddressType, InternalAddressType or AddressType). The thing to consider now is, how will the XML parser know which type you meant to use, surely it needs to know otherwise it can not do proper validation?

Well, it knows because if you want to use a type other than the one explicitly specified in the schema (in this case "AddressType") then you have to let the parser know which type your using. This is done in the XML document using the **xsi:type** attribute.

**Let's look at an example:**

*<xs:element name="Person">*

*<xs:complexType>*

*<xs:sequence>*

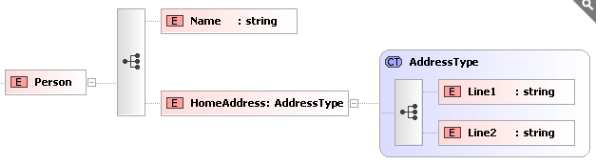
*<xs:element name="Name" type="xs:string" />*

*<xs:element name="HomeAddress" type="AddressType" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*



**Sample XML for the above may look like the following:**

*<?xml version="1.0" ?>*

*<Person>*

*<Name>Fred</Name>*

*<HomeAddress>*

*<Line1>22 whatever place, someplace</Line1>*

*<Line2>sometown, ss1 6gy </Line2>*

*</HomeAddress>*

*</Person>*

**However, the following is also valid:**

*<?xml version="1.0" ?>*

*<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">*

*<Name>Fred</Name>*

*<HomeAddress xsi:type="USAddressType">*

*<Line1>234 Lancaseter Av</Line1>*

*<Line2>SmallsVille</Line2>*

*<State>Florida</State>*

*<Zipcode>34543</Zipcode>*

*</HomeAddress>*

*</Person>*

**Let's look at that in more detail.**

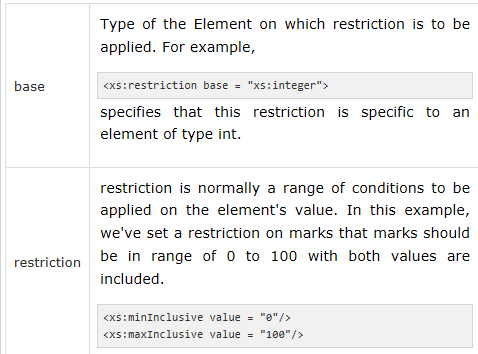
* We have added the attribute xsi:type="USAddressType" to the "HomeAddress" element. This tells the XML parser that the element actually contains data described by "USAddressType".
* The xmlns:xsi attribute in the root element (Person) tells the XML parser that the alias xsi maps to the namespace "http://www.w3.org/2001/XMLSchema-instance".
* The xsi: part of the xsi:type attribute is a namespace qualifier. It basically says the attribute "type" is from the namespace that is aliased by "xsi" which was defined earlier to mean "http://www.w3.org/2001/XMLSchema-instance".
* The "type" attribute in this namespace is an instruction to the XML Parser to tell it which definition to use to validate the element.

**XSD – Restriction**

Restriction element is used to define accepted values that an XML element can take.

**Syntax**

*<xs:restriction base = "element-type"> restrictions </xs:restriction>*



**Examples :**

1. **Restriction on Value.**

**Condition** – “Marks should be in range of 0 to 100”.

*<xs:element name = "marks">*

*<xs:simpleType>*

*<xs:restriction base = "xs:integer">*

*<xs:minInclusive value = "0"/>*

*<xs:maxInclusive value = "100"/>*

*</xs:restriction>*

*</xs:simpleType>*

*</xs:element>*

1. **Restriction on Set of Values.**

**Condition** – “Grades should only be A, B or C”.

*<xs:element name = "grades">*

*<xs:simpleType>*

*<xs:restriction base = "xs:string">*

*<xs:enumeration value = "A"/>*

*<xs:enumeration value = "B"/>*

*<xs:enumeration value = "C"/>*

*</xs:restriction>*

*</xs:simpleType>*

*</xs:element>*

1. **Restriction using regular pattern**.

**Condition** − firstname should be in alphabets only.

*<xs:element name = "firstname">*

*<xs:simpleType>*

*<xs:restriction base = "xs:string">*

*<xs:pattern value = "[a-z]"/>*

*</xs:restriction>*

*</xs:simpleType>*

*</xs:element>*

**Types of Restrictions**

|  |  |
| --- | --- |
|  | **Enumeration**  Defines a list of values which are acceptable. |
|  | **fractionDigits**  Defines the maximum number of decimal places allowed (zero or more). |
|  | **Length**  Defines length in terms of characters of string or items in a list (zero or more). |
|  | **maxExclusive**  Defines upper bounds for numeric values excluding this number. |
|  | **maxInclusive**  Defines upper bounds for numeric values including this number. |
|  | **maxLength**  Defines maximum length in terms of characters of string or items in a list (zero or more). |
|  | **minExclusive**  Defines lower bounds for numeric values excluding this number. |
|  | **minInclusive**  Defines lower bounds for numeric values including this number. |
|  | **minLength**  Defines minimum length in terms of characters of string or items in a list (zero or more). |
|  | **Pattern**  Defines the exact sequence of characters identified by the pattern that are acceptable |
|  | **totalDigits**  Defines the exact number of digits allowed in the number (always greater than zero) |
|  | **whitespace**  Defines the way in which white space characters (line feeds, tabs, spaces, and carriage returns) are handled |

## ****Extending**** Simple Types

There are 3 ways in which a simpleType can be extended; Restriction, List and Union. The most common is Restriction, but we will cover the other 2 as well.

### Restriction

Restriction is a way to constrain an existing type definition. We can apply a restriction to the built in data types xs:string, xs:integer, xs:date, etc. or ones we create ourselves.

Here we are defining a restriction the existing type "string", we are applying a regular expression to it, to limit the values it can take.

*<xs:simpleType name="LetterType">*

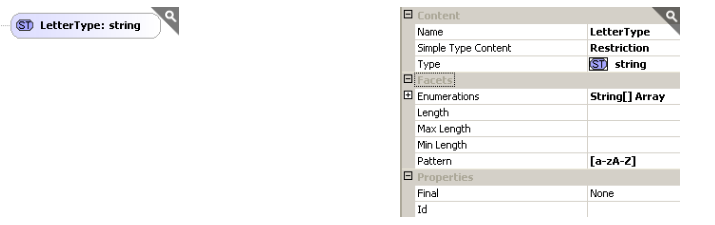
*<xs:restriction base="xs:string">*

*<xs:pattern value="[a-zA-Z]" />*

*</xs:restriction>*

*</xs:simpleType>*

**This can be shown graphically in Liquid XML Studio as follows:**



**Let's go through this line by line.**

1. A <simpleTyp> tag is used to define a our new type, we must give the type a unique name - in this case "LetterType".

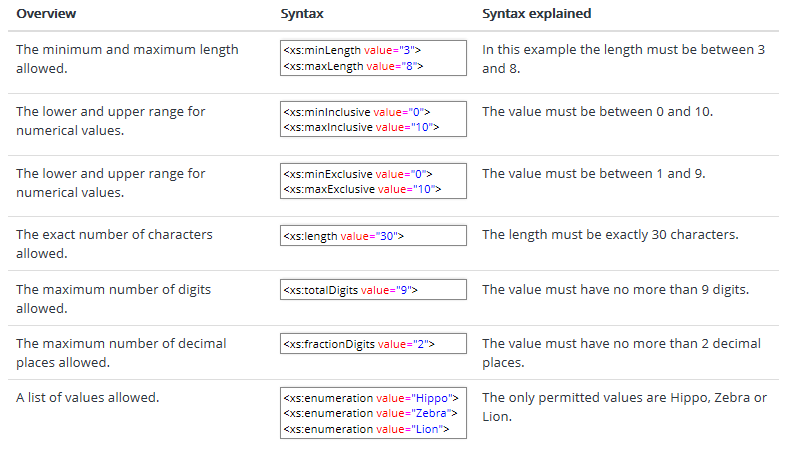
2. We are restricting an existing type - so the tag is <restriction> (you can also extend an existing type - but more about this later). We are basing our new type on a string so type="xs:string".

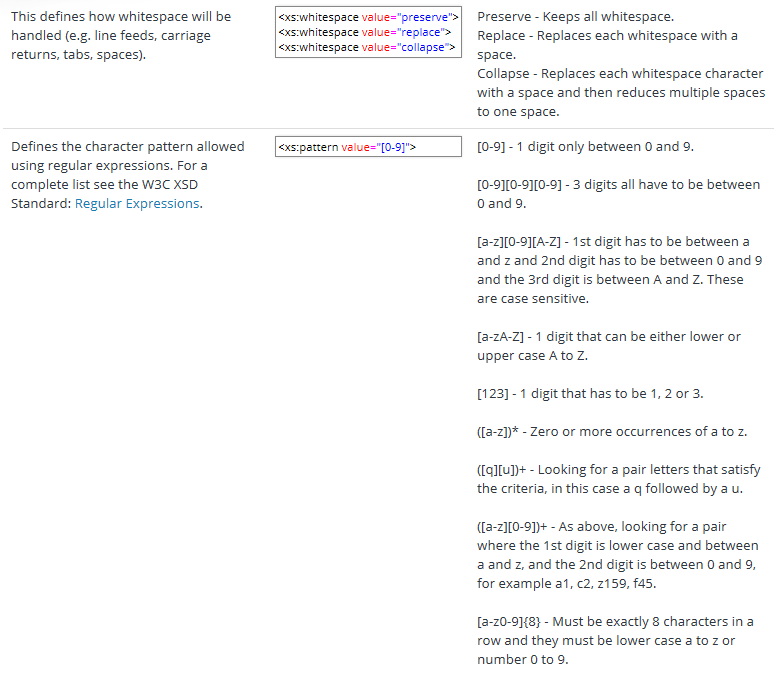
3. We are applying a restriction in the form of a Regular expression, this is specified using the <pattern> element. The regular expression means the data must contain a single lower or upper case letter a through to z.

4. closing tag for the restriction.

5. closing tag for the simple type.

Restrictions may also be referred to as **Facets**.





It is important to note that not all facets are valid for all data types - for example, maxInclusive has no meaning when applied to a string. For the combinations of facets that are valid for a given data type refer to the W3C XSD standard.

## Restrictions on Values

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

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

## Restrictions on a Set of Values

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

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

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

The example above could also have been written like this:

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

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

## Restrictions on a Series of Values

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

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

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

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

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

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

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

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

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

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

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

## Other Restrictions on a Series of Values

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

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

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

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

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

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

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

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

## Restrictions on Whitespace Characters

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

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

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

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

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

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

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

## Restrictions on Length

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

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

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

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

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

## Restrictions for Datatypes

|  |  |
| --- | --- |
| **Constraint** | **Description** |
| enumeration | Defines a list of acceptable values |
| fractionDigits | Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero |
| Length | Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero |
| maxExclusive | Specifies the upper bounds for numeric values (the value must be less than this value) |
| maxInclusive | Specifies the upper bounds for numeric values (the value must be less than or equal to this value) |
| maxLength | Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero |
| minExclusive | Specifies the lower bounds for numeric values (the value must be greater than this value) |
| minInclusive | Specifies the lower bounds for numeric values (the value must be greater than or equal to this value) |
| minLength | Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero |
| Pattern | Defines the exact sequence of characters that are acceptable |
| totalDigits | Specifies the exact number of digits allowed. Must be greater than zero |
| whiteSpace | Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled |

**XSD - Complex Empty Element**

Complex Empty Element can only have attribute, but no content.

**See the following example −**

***<student rollno = "393" />***

**We can declare Complex Empty elements using the following methods −**

1. **Use type attribute**

Define a complex type element "**StudentType**" and then create element student of type "**StudentType**".

*<xs:complexType name = "StudentType">*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

1. **Use ComplexContent**

Define an element of complexType with complexContent. ComplexContent specifies that the content of the element is to be restricted.

*<xs:element name = "student">*

*<xs:complexType>*

*<xs:complexContent>*

*<xs:restriction base = "xs:integer">*

*<xs:attribute name = "rollno" type = "xs:positiveInteger"/>*

*</xs:restriction>*

*</xs:complexContent>*

*</xs:complexType>*

*</xs:element>*

1. **Use ComplexType alone**

Define an element of complexType with required attribute element only.

*<xs:element name = "student">*

*<xs:complexType>*

*<xs:attribute name = "rollno" type = "xs:positiveInteger"/>*

*</xs:complexType>*

*</xs:element>*

**XSD - Complex Element Only**

Complex Elements only can only have other elements. See the following example −

*<student>*

*<firstname>Vaneet</firstname>*

*<lastname>Gupta</lastname>*

*<nickname>Vinni</nickname>*

*<marks>95</marks>*

*</student>*

We can declare Complex element-only element using the following methods −

1. **Use type attribute**

Define a complex type element "StudentType" and then create an element called **student** of 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 = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:string"/>*

*</xs:sequence>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

In the above example, we've used **sequence**. It is used to maintain the order in which the elements are to be present in the XML. If order is not maintained, then XML will not be validated.

1. **Use ComplexType alone**

Define an element of complexType with the required attribute element only.

*<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 = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:string"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*</xs:element>*

**XSD - Complex Text Only Element**

Complex Text-only Element can only have text and attribute, but no content.

**See the following example −**

***<marks grade = "A" >90</student>***

We can declare Complex Text-only elements using the following methods −

1. **Use SimpleContent**

Define complexType with simpleContent. SimpleContent can use extension/restriction element to increase/reduce scope of base type of the element. Create an element of defined complexType using **type** attribute.

*<xs:element name = "marks" type = "marksType"/>*

*<xs:complexType name = "marksType">*

*<xs:simpleContent>*

*<xs:extension base = "xs:integer">*

*<xs:attribute name = "grade" type = "xs:string" />*

*</xs:extension>*

*</xs:simpleContent>*

*</xs:complexType>*

1. **Use ComplexType alone**

Define an element of complexType with the required attribute element only.

*<xs:element name = "marks">*

*<xs:complexType>*

*<xs:simpleContent>*

*<xs:extension base = "xs:integer">*

*<xs:attribute name = "grade" type = "xs:string" />*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

*</xs:element>*

**XSD - Complex Mix Element**

Complex Mix Element can have text and attribute and elements.

**See the following example −**

*<student rollno = "393">*

*Dear <firstname>Dinkar</firstname>*

*<lastname>Kad</lastname>*

*<nickname>Dinkar</nickname>*

*<marks>85</marks>*

*</student>*

**We can declare such Complex Text using the following ways −**

1. **Use mixed=true**

Define complexType with attribute "mixed" set to true. "mixed" attribute allow to have character data between elements.

*<xs:complexType name = "StudentType" mixed = "true">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type="xs:positiveInteger"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*<xs:element name = 'student' type = 'StudentType' />*

1. **Use ComplexType alone**

Define an element of complexType with the required attribute element only.

*<xs:element name = 'student'>*

*<xs:complexType mixed = "true">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:string"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*</xs:element>*

### Union

A union is a mechanism for combining two or more different data types into one.

The following defines two simple types "SizeByNumberType" all the positive integers up to 21 (e.g. 10, 12, 14), and "SizeByStringNameType" the values small, medium and large.

*<xs:simpleType name="SizeByNumberType">*

*<xs:restriction base="xs:positiveInteger">*

*<xs:maxInclusive value="21" />*

*</xs:restriction>*

*</xs:simpleType>*

*<xs:simpleType name="SizeByStringNameType">*

*<xs:restriction base="xs:string">*

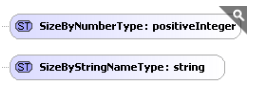
*<xs:enumeration value="small" />*

*<xs:enumeration value="medium" />*

*<xs:enumeration value="large" />*

*</xs:restriction>*

*</xs:simpleType>*

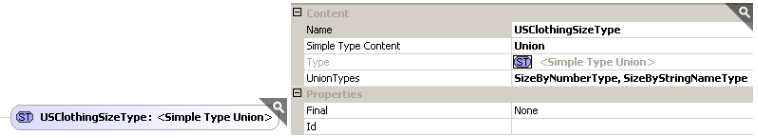


We can then define a new type called "USClothingSizeType", we define this as a union of the types "SizeByNumberType" and "SizeByStringNameType" (although we can add any number of types, including the built in types - separated by whitespace).

*<xs:simpleType name="USClothingSizeType">*

*<xs:union memberTypes="SizeByNumberType SizeByStringNameType" />*

*</xs:simpleType>*



This means the type can contain any of the values that the two members can take (e.g. 1, 2, 3, ...., 20, 21, small, medium, large).

This new type can then be used in the same way as any other <xs:simpleType>.

### List

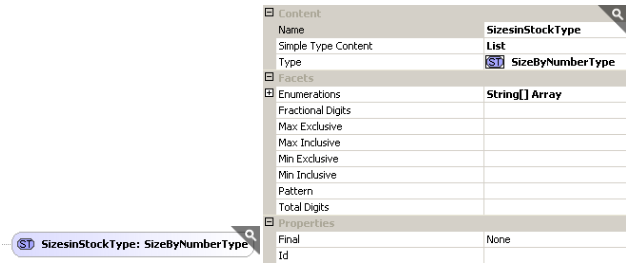
A list allows the value (in the XML document) to contain a number of valid values separated by whitespace.

A List is constructed in a similar way to a Union. The difference being that we can only specify a single type. This new type can contain a list of values that are defined by the itemType property. The values must be whitespace separated. So a valid value for this type would be "5 9 21".

*<xs:simpleType name="SizesinStockType">*

*<xs:list itemType="SizeByNumberType" />*

*</xs:simpleType>*



## ****Namespaces**** Overview

In Java, packages are meant for avoiding the naming collisions that occur between classes provided by multiple developers.

Similarly to avoid the naming collisions that occur between different user defined elements and complex types. So, we can group them under one umbrella / group / compartmentaize, which are called as Namespaces.

If you are creating and modifying XML documents validating against XML Schema making use of namespaces, then [XML Data Binding](https://www.liquid-technologies.com/xml-data-binder) will save you a great deal of time as mostly removes this complexity.

Namespaces are a mechanism for breaking up your schemas. Up until now we have assumed that you only have a single schema file containing all your element definitions, but the XSD standard allows you to structure your XSD schemas by breaking them into multiple files. These child schemas can then be included into a parent schema.

Breaking schemas into multiple files can have several advantages. You can create re-usable definitions that can be used across several projects. They make definitions easier to read and version as they break down the schema into smaller units that are simpler to manage.

## Namespace Walk-through ****Example****

**In this example, the schema is broken out into four files:**

* CommonTypes - this could contain all your basic types such as AddressType, PriceType, and PaymentMethodType
* CustomerTypes - this could contain all your definitions for your customers.
* OrderTypes - this could contain all your definitions for orders.
* Main - this would pull all the sub schemas together into a single schema, and define your main elements.

This all works fine without namespaces, but if different teams start working on different files, then you have the possibility of name clashes, and it would not always be obvious where a definition had come from. The solution is to place the definitions for each schema file within a distinct namespace.

**We can do this by adding the attribute targetNamespace into the schema element in the XSD file:**

*<?xml version="1.0" ?>*

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

*targetNamespace="myNamespace">*

*...*

*</xs:schema>*

The value of targetNamespace is simply a unique identifier, typically a company may use their URL followed by something descriptive to qualify it.

In principle the namespace has no meaning, but some companies have used the URL where the schema is stored as the targetNamespace, and so some XML parsers will use this as a hint path for the schema:

**targetNamespace="http://www.microsoft.com/CommonTypes.xsd"**

**However, the following would be equally valid:**

targetNamespace="my-common-types"

Placing the targetNamespace attribute at the top of your XSD schema means that all entities defined in it are part of this namespace. So in our example above each of the 4 schema files could have a distinct targetNamespace value.

**Let's look at them in detail.**

### CommonTypes.xsd

*<?xml version="1.0" encoding="utf-16" ?>*

*<xs:schema targetNamespace="http://NamespaceTest.com/CommonTypes"*

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

*elementFormDefault="qualified">*

*<xs:complexType name="AddressType">*

*<xs:sequence>*

*<xs:element name="Line1" type="xs:string" />*

*<xs:element name="Line2" type="xs:string" />*

*</xs:sequence>*

*</xs:complexType>*

*<xs:simpleType name="PriceType">*

*<xs:restriction base="xs:decimal">*

*<xs:fractionDigits value="2" />*

*</xs:restriction>*

*</xs:simpleType>*

*<xs:simpleType name="PaymentMethodType">*

*<xs:restriction base="xs:string">*

*<xs:enumeration value="VISA" />*

*<xs:enumeration value="MasterCard" />*

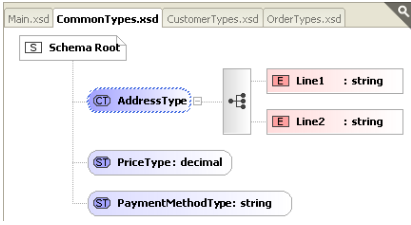
*<xs:enumeration value="Cash" />*

*<xs:enumeration value="AMEX" />*

*</xs:restriction>*

*</xs:simpleType>*

*</xs:schema>*



This schema defines some basic re-usable entities and types. The use of the targetNamespace attribute in the <xs:schema> element ensures all the enclosed definitions (AddressType, PriceType and PaymentMethodType) are all in the namespace:

targetNamespace="http://NamespaceTest.com/CommonTypes"

### CustomerTypes.xsd

*<?xml version="1.0" encoding="utf-16" ?>*

*<xs:schema xmlns:cmn="http://NamespaceTest.com/CommonTypes"*

*targetNamespace="http://NamespaceTest.com/CustomerTypes"*

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

*elementFormDefault="qualified">*

*<xs:import schemaLocation="CommonTypes.xsd"*

*namespace="http://NamespaceTest.com/CommonTypes" />*

*<xs:complexType name="CustomerType">*

*<xs:sequence>*

*<xs:element name="Name" type="xs:string" />*

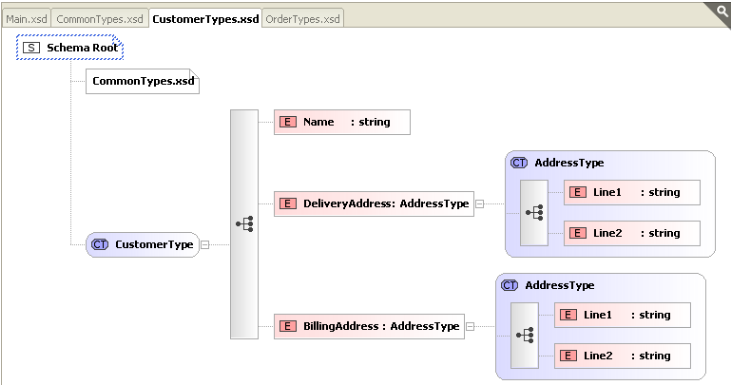
*<xs:element name="DeliveryAddress" type="cmn:AddressType" />*

*<xs:element name="BillingAddress" type="cmn:AddressType" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:schema>*



This schema defines the entity CustomerType, which makes use of the AddressType defined in the CommonTypes.xsd schema. We need to do a few things in order to use this.

First we need to import that schema into this one - so we can see it. This is done using <xs:import>. It is worth noting the presence of the targetNamespace attribute at this point. This means that all entities defined in this schema belong to that namespace:

***targetNamespace="http://NamespaceTest.com/CustomerTypes"***

So in order to make use of the AddressType which is defined in CommonTypes.xsd, and part of the namespace "http://NamespaceTest.com/CommonTypes", we must fully qualify it.

In order to do this we must define an alias for the namespace "http://NamespaceTest.com/CommonTypes", we do this by adding another attribute in the <xs:schema> element:

**xmlns:cmn="http://NamespaceTest.com/CommonTypes"**

This specifies that the alias "cmn" represents the namespace "http://NamespaceTest.com/CommonTypes". We can now make use of the types within the CommonTypes.xsd schema. When we do this we must fully qualify them as they are not in the same targetNamespace as the schema that is using them:

*<xs:element name="DeliveryAddress" type="cmn:AddressType" />*

*<xs:element name="BillingAddress" type="cmn:AddressType" />*

### OrderType.xsd

*<?xml version="1.0" encoding="utf-16" ?>*

*<xs:schema xmlns:cmn="http://NamespaceTest.com/CommonTypes"*

*targetNamespace="http://NamespaceTest.com/OrderTypes"*

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

*elementFormDefault="qualified">*

*<xs:import schemaLocation="CommonTypes.xsd"*

*namespace="http://NamespaceTest.com/CommonTypes" />*

*<xs:complexType name="OrderType">*

*<xs:sequence>*

*<xs:element maxOccurs="unbounded" name="Item">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="ProductName" type="xs:string" />*

*<xs:element name="Quantity" type="xs:int" />*

*<xs:element name="UnitPrice" type="cmn:PriceType" />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:schema>*

This schema defines the type OrderType which is within the namespace "http://NamespaceTest.com/OrderTypes":

**targetNamespace="http://NamespaceTest.com/OrderTypes"**

The constructs used here are the same as those used in CustomerTypes.xsd.

### Main.xsd

*<?xml version="1.0" encoding="utf-16" ?>*

*<xs:schema xmlns:ord="http://NamespaceTest.com/OrderTypes"*

*xmlns:pur="http://NamespaceTest.com/Purchase"*

*xmlns:cmn="http://NamespaceTest.com/CommonTypes"*

*xmlns:cust="http://NamespaceTest.com/CustomerTypes"*

*targetNamespace="http://NamespaceTest.com/Purchase"*

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

*elementFormDefault="qualified">*

*<xs:import schemaLocation="CommonTypes.xsd"*

*namespace="http://NamespaceTest.com/CommonTypes" />*

*<xs:import schemaLocation="CustomerTypes.xsd"*

*namespace="http://NamespaceTest.com/CustomerTypes" />*

*<xs:import schemaLocation="OrderTypes.xsd"*

*namespace="http://NamespaceTest.com/OrderTypes" />*

*<xs:element name="Purchase">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="OrderDetail" type="ord:OrderType" />*

*<xs:element name="PaymentMethod" type="cmn:PaymentMethodType" />*

*<xs:element ref="pur:CustomerDetails" />*

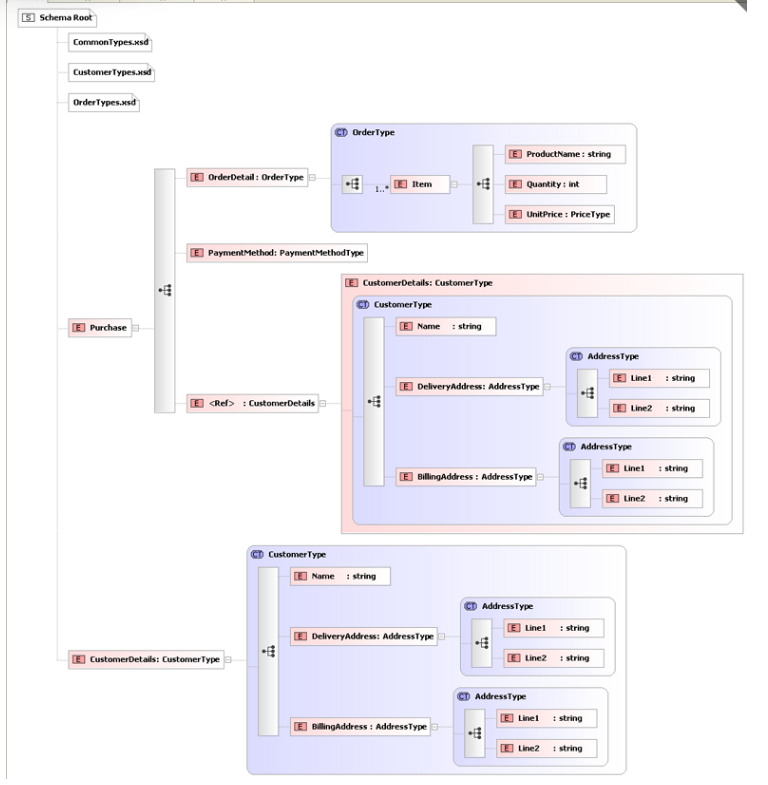
*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:element name="CustomerDetails" type="cust:CustomerType" />*

*</xs:schema>*



The elements in this schema are part of the namespace "http://NamespaceTest.com/Purchase":

*targetNamespace="http://NamespaceTest.com/Purchase"*

This is our main schema and defines the concrete elements "Purchase", and "CustomerDetails". This element builds on the other schemas, so we need to import them all, and define aliases for each namespace.

**Note**: The element "CustomerDetails" which is defined in main.xsd is referenced from within "Purchase".

### XML Document

As the root element "Purchase" is in the namespace "http://NamespaceTest.com/Purchase", we must quantify the <Purchase> element within the resulting XML document. Let's look at an example:

*<?xml version="1.0" ?>*

*<p:Purchase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://NamespaceTest.com/Purchase Main.xsd"*

*xmlns:p="http://NamespaceTest.com/Purchase"*

*xmlns:o="http://NamespaceTest.com/OrderTypes"*

*xmlns:c="http://NamespaceTest.com/CustomerTypes"*

*xmlns:cmn="http://NamespaceTest.com/CommonTypes">*

*<p:OrderDetail>*

*<o:Item>*

*<o:ProductName>Widget</o:ProductName>*

*<o:Quantity>1</o:Quantity>*

*<o:UnitPrice>3.42</o:UnitPrice>*

*</o:Item>*

*</p:OrderDetail>*

*<p:PaymentMethod>VISA</p:PaymentMethod>*

*<p:CustomerDetails>*

*<c:Name>James</c:Name>*

*<c:DeliveryAddress>*

*<cmn:Line1>15 Some Road</cmn:Line1>*

*<cmn:Line2>SomeTown</cmn:Line2>*

*</c:DeliveryAddress>*

*<c:BillingAddress>*

*<cmn:Line1>15 Some Road</cmn:Line1>*

*<cmn:Line2>SomeTown</cmn:Line2>*

*</c:BillingAddress>*

*</p:CustomerDetails>*

*</p:Purchase>*

The first thing we see is the xsi:schemaLocation attribute in the root element. This tells the XML parser that elements within the namespace "http://NamespaceTest.com/Purchase" can be found in the file "Main.xsd" (Note: the namespace and URL are separated with whitespace, such as a carriage return or space).

**The next thing we do is define some aliases:**

* "p" to mean the namespace "http://NamespaceTest.com/Purchase"
* "c" to mean the namespace "http://NamespaceTest.com/CustomerTypes"
* "o" to mean the namespace "http://NamespaceTest.com/OrderTypes"
* "cmn" to mean the namespace "http://NamespaceTest.com/CommonTypes"

You have probably noticed that every element in the schema is qualified with one of these aliases.

**The general rules for this are:**

The alias must be the same as the target namespace in which the element is defined. It is important to note that this is where the element is defined - not where the complexType is defined.

So the element <OrderDetail> is actually defined in main.xsd and so it is part of the namespace "http://NamespaceTest.com/Purchase", even though it uses the complexType "OrderType" which is defined in the OrderTypes.xsd. The contents of <OrderDetail> are defined within the complexType "OrderType", which is in the target namespace "http://NamespaceTest.com/OrderTypes", so the child element <Item> needs qualifying within the namespace "http://NamespaceTest.com/OrderTypes".

### The elementFormDefault Attribute

You may have noticed that each schema contained an attribute elementFormDefault="qualified". This has two possible values, qualified, and unqualified, the default is unqualified. This attribute changes the namespace rules considerably. It is normally easier to set it to qualified.

So to see the effects of this property, if we set it to be unqualified in all of our schemas, the resulting XML would look like this:

*<?xml version="1.0" ?>*

*<p:Purchase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://NamespaceTest.com/Purchase Main.xsd"*

*xmlns:p="http://NamespaceTest.com/Purchase">*

*<OrderDetail>*

*<Item>*

*<ProductName>Widget</ProductName>*

*<Quantity>1</Quantity>*

*<UnitPrice>3.42</UnitPrice>*

*</Item>*

*</OrderDetail>*

*<PaymentMethod>VISA</PaymentMethod>*

*<p:CustomerDetails>*

*<Name>James</Name>*

*<DeliveryAddress>*

*<Line1>15 Some Road</Line1>*

*<Line2>SomeTown</Line2>*

*</DeliveryAddress>*

*<BillingAddress>*

*<Line1>15 Some Road</Line1>*

*<Line2>SomeTown</Line2>*

*</BillingAddress>*

*</p:CustomerDetails>*

*</p:Purchase>*

**This is considerably different from the previous XML document. These general rules now apply:**

* Only root elements defined within a schema need qualifying with a namespace.
* All types that are defined inline do NOT need to be qualified.

The first element is Purchase, this is defined globally in the Main.xsd schema, and therefore needs qualifying within the schemas target namespace "http://NamespaceTest.com/Purchase".

The first child element is <OrderDetail> and is defined inline in Main.xsd->Purchase. So it does not need to be aliased.

The same is true for all the child elements, they are all defined inline, so they do not need qualifying with a namespace.

The final child element <CustomerDetails> is a little different. As you can see we have defined this as a global element within the targetNamespace "http://NamespaceTest.com/Purchase". In the element "Purchase" we just reference it. Because we are using a reference to an element, we must take into account its namespace, thus we alias it <p:CustomerDetails>.

## Element and Attribute ****Groups****

Elements and Attributes can be grouped together using <xs:group> and <xs:attributeGroup>. These groups can then be referred to elsewhere within the schema. Groups must have a unique name and be defined as children of the <xs:schema> element. When a group is referred to, it is as if its contents have been copied into the location it is referenced from.

Note:

<xs:group> and <xs:attributeGroup> can not be extended or restricted in the way <xs:complexType> or <xs:simpleType> can. They are purely to group a number of items of data that are always used together. For this reason they are not the first choice of constructs for building reusable maintainable schemas, but they can have their uses.

*<xs:group name="CustomerDataGroup">*

*<xs:sequence>*

*<xs:element name="Forename" type="xs:string" />*

*<xs:element name="Surname" type="xs:string" />*

*<xs:element name="Dob" type="xs:date" />*

*</xs:sequence>*

*</xs:group>*

*<xs:attributeGroup name="DobPropertiesGroup">*

*<xs:attribute name="Day" type="xs:string" />*

*<xs:attribute name="Month" type="xs:string" />*

*<xs:attribute name="Year" type="xs:integer" />*

*</xs:attributeGroup>*

**These groups can then be referenced in the definition of complex types, as shown below:**

*<xs:complexType name="Customer">*

*<xs:sequence>*

*<xs:group ref="CustomerDataGroup" />*

*<xs:element name="..." type="..." />*

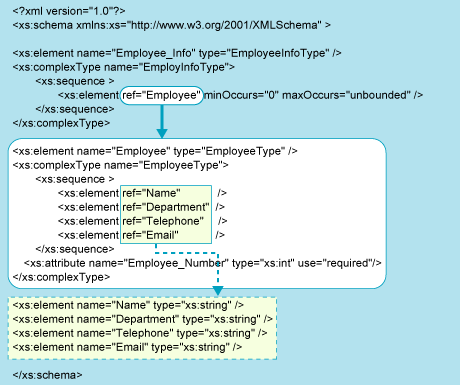
*</xs:sequence>*

*<xs:attributeGroup ref="DobPropertiesGroup" />*

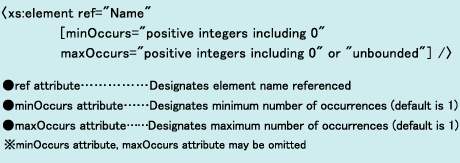
*</xs:complexType>*

## Example:

**Element Declaration Reference for a Model Group Element**



The element reference syntax is as follows:



## The <any> Element

The <any> construct allows us specify that our XML document can contain elements that are not defined in this schema. A typical use for this is when you define a message envelope. For example, the message payload is unknown to the system, but we can still validate the message.

**Look at the following schema:**

*<xs:element name="Message">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name="DateSent" type="xs:date" />*

*<xs:element name="Sender" type="xs:string" />*

*<xs:element name="Content">*

*<xs:complexType>*

*<xs:sequence>*

*<xs:any />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

We have defined an element called "Message", which must have a "DateSent" child element (which is a date), a "Sender" child element (which must be a string), and a "Content" child element - which can contain any element - it doesn't even have to be described in the schema.

**So the following XML would be acceptable.**

*<Message>*

*<DateSent>2000-01-12</DateSent>*

*<Sender>Admin</Sender>*

*<Content>*

*<AccountCreationRequest>*

*<AccountName>Fred</AccountName>*

*</AccountCreationRequest>*

*</Content>*

*</Message>*

The <any> construct has a number of properties that can further restrict what can be used in its place.

minOccurs and maxOccurs allows you to specify how may instances of undefined elements must be placed within the XML document.

namespace allows you to specify that the undefined element must belong to a given namespace. This may be a list of namespace's (space separated). There are also three built in values ##any, ##other, ##targetnamespace, ##local. Consult the [XSD standard](http://www.w3.org/TR/xmlschema-0/#any) for more information on this.

**processContents tells the XML parser how to deal with the unknown elements. The values are:**

* **Skip** - no validation is performed - but it must be well formed XML.
* **Lax** - if there is a schema to validate the element, then it must be valid against it, if there is no schema, then that's OK.
* **Strict** - There must be a definition for the element available to the parser, and it must be valid against it.

**RECAP:**

<any> element is used to extend the XSD functionality. It is used to extend a complexType element defined in one XSD by an element which is not defined in the schema.

Consider an example − person.xsd has defined **person** complexType element. address.xsd has defined **address** complexType element.

**person.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:element name = "person">

<xs:complexType >

<xs:sequence>

<xs:element name = "firstname" type = "xs:string"/>

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

<xs:element name = "nickname" type = "xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

**address.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:element name = "address">

<xs:complexType>

<xs:sequence>

<xs:element name = "houseNumber" type = "xs:string"/>

<xs:element name = "street" type = "xs:string"/>

<xs:element name = "state" type = "xs:string"/>

<xs:element name = "zipcode" type = "xs:integer"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

If we want to define a person with address in XML, then the following declaration will be invalid.

**person.xml**

<?xml version = "1.0"?>

<class xmlns = "http://www.tutorialspoint.com"

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

xsi:schemaLocation = "http://www.tutorialspoint.com person.xsd

http://www.tutorialspoint.com address.xsd">

<person>

<firstname>Dinkar</firstname>

<lastname>Kad</lastname>

<nickname>Dinkar</lastname>

<address>

<houseNumber>101</firstname>

<street>Sector-1,Patiala</lastname>

<state>Punjab</lastname>

<zipcode>301202<zipcode>

</address>

</person>

</class>

**Use <xs:any>**

In order to validate above person.xml, add <xs:any> to person element in person.xsd.

**person.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:element name = "person">

<xs:complexType >

<xs:sequence>

<xs:element name = "firstname" type = "xs:string"/>

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

<xs:element name = "nickname" type = "xs:string"/>

<xs:any minOccurs = "0"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

Now person.xml will be validated against person.xsd and address.xsd.

## The <anyAttribute> Element

<anyAttribute> works in the same way as <any>, except it allows unknown attributes to be inserted into a given element.

*<xs:element name="Sender">*

*<xs:complexType>*

*<xs:simpleContent>*

*<xs:extension base="xs:string">*

*<xs:anyAttribute />*

*</xs:extension>*

*</xs:simpleContent>*

*</xs:complexType>*

*</xs:element>*

**This would mean that we can add any attributes we like to the Sender element, and the XML document would still be valid:**

*<Sender ID="7687">Fred</Sender>*

<xs:anyAttribute> element is used to extend the XSD functionality. It is used to extend a complexType element defined in one xsd by an attribute which is not defined in the schema.

Consider an example − person.xsd has defined **person** complexType element. attributes.xsd has defined **age** attribute.

**person.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:element name = "person">

<xs:complexType >

<xs:sequence>

<xs:element name = "firstname" type = "xs:string"/>

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

<xs:element name = "nickname" type = "xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

**attributes.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:attribute name = "age">

<xs:simpleType>

<xs:restriction base = "xs:integer">

<xs:pattern value = "[0-100]"/>

</xs:restriction>

</xs:simpleType>

</xs:attribute>

</xs:schema>

If we want to define a person with age in XML, then the following declaration will be invalid.

**person.xml**

<?xml version = "1.0"?>

<class xmlns = "http://www.tutorialspoint.com"

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

xsi:schemaLocation = "http://www.tutorialspoint.com person.xsd

http://www.tutorialspoint.com attributes.xsd">

<person age = "20">

<firstname>Dinkar</firstname>

<lastname>Kad</lastname>

<nickname>Dinkar</lastname>

</person>

</class>

Use <xs:anyAttribute>

In order to validate above **person.xml**, add <xs:anyAttribute> to **person** element in person.xsd.

**person.xsd**

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

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

targetNamespace = "http://www.tutorialspoint.com"

xmlns = "http://www.tutorialspoint.com"

elementFormDefault = "qualified">

<xs:element name = "person">

<xs:complexType >

<xs:sequence>

<xs:element name = "firstname" type = "xs:string"/>

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

<xs:element name = "nickname" type = "xs:string"/>

</xs:sequence>

<xs:anyAttribute/>

</xs:complexType>

</xs:element>

</xs:schema>

Now **person.xml** will be validated against **person.xsd** and **attributes.xsd**.

**Note:**

|  |  |  |
| --- | --- | --- |
| Meaning of the Model Group | XML Schema | DTD |
| Output the element in the written order in the exact number of occurrences designated | sequence element | ， |
| Output any one element in the exact number of occurrences designated | choice element | ｜ |

**XML Schema and DTD Differences related to Attribute Declarations**

|  |  |  |
| --- | --- | --- |
| Designations related to Occurrences | XML Schema | DTD |
| Attribute description may be omitted | optional | #IMPLIED |
| Attribute description is required | required | #REQUIRED |
| Attribute description is prohibited | prohibited | None |

## XML Schemas Secure Data Communication

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

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

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

However, an XML element with a data type like this:

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

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

## Well-Formed is Not Enough

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

* it must begin with the XML declaration
* it must have one unique root element
* start-tags must have matching end-tags
* elements are case sensitive
* all elements must be closed
* all elements must be properly nested
* all attribute values must be quoted
* entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.

Think of the following situation: you order 5 gross of laser printers, instead of 5 laser printers. With XML Schemas, most of these errors can be caught by your validating software.

## Occurrence Indicators

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

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

### maxOccurs Indicator

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

*<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="full\_name" type="xs:string"/>  
      <xs:element name="child\_name" type="xs:string" maxOccurs="10"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>*

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

### minOccurs Indicator

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

*<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="full\_name" type="xs:string"/>  
      <xs:element name="child\_name" type="xs:string"  
      maxOccurs="10" minOccurs="0"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>*

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

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

## Group Indicators

Group indicators are used to define related sets of elements.

### Element Groups

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

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

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

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

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

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

### Attribute Groups

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

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

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

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

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

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

# XSD The <anyAttribute> Element

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

## The <anyAttribute> Element

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

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

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

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

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

*<?xml version="1.0" encoding="UTF-8"?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
targetNamespace="http://www.w3schools.com"  
xmlns="http://www.w3schools.com"  
elementFormDefault="qualified">*  
*<xs:attribute name="gender">  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="male|female"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:attribute>  
  
</xs:schema>*

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

*<?xml version="1.0" encoding="UTF-8"?>  
  
<persons xmlns="http://www.microsoft.com"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:SchemaLocation="http://www.microsoft.com family.xsd  
http://www.w3schools.com attribute.xsd">  
  
<person gender="female">  
  <firstname>Hege</firstname>  
  <lastname>Refsnes</lastname>  
</person>  
  
<person gender="male">  
  <firstname>Stale</firstname>  
  <lastname>Refsnes</lastname>  
</person>  
  
</persons>*

The XML file above is valid because the schema "family.xsd" allows us to add an attribute to the "person" element.

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

# XSD Element Substitution

With XML Schemas, one element can substitute another element.

## Element Substitution

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

To solve this problem, we could define a **substitutionGroup** in the XML schema. First, we declare a head element and then we declare the other elements which state that they are substitutable for the head element.

*<xs:element name="name" type="xs:string"/>  
<xs:element name="navn" substitutionGroup="name"/>*

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

**Look at this fragment of an XML schema:**

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

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

*<customer>  
  <name>John Smith</name>  
</customer>*

or like this:

*<kunde>  
  <navn>John Smith</navn>  
</kunde>*

## Blocking Element Substitution

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

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

**Look at this fragment of an XML schema:**

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

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

*<customer>  
  <name>John Smith</name>  
</customer>*

**BUT THIS IS NO LONGER VALID:**

*<kunde>  
  <navn>John Smith</navn>  
</kunde>*

## Using substitutionGroup

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

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

## What are Global Elements?

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

## An XML Document

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

*<?xml version="1.0" encoding="UTF-8"?>  
  
<shiporder orderid="889923"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:noNamespaceSchemaLocation="shiporder.xsd">  
  <orderperson>John Smith</orderperson>  
  <shipto>  
    <name>Ola Nordmann</name>  
    <address>Langgt 23</address>  
    <city>4000 Stavanger</city>  
    <country>Norway</country>  
  </shipto>  
  <item>  
    <title>Empire Burlesque</title>  
    <note>Special Edition</note>  
    <quantity>1</quantity>  
    <price>10.90</price>  
  </item>  
  <item>  
    <title>Hide your heart</title>  
    <quantity>1</quantity>  
    <price>9.90</price>  
  </item>  
</shiporder>*

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

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

## Create an XML Schema

Now we want to create a schema for the XML document above.

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

<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
...  
</xs:schema>

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

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

*<xs:element name="shiporder">  
  <xs:complexType>  
    <xs:sequence>  
      ...  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>*

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

*<xs:element name="orderperson" type="xs:string"/>*

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

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

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

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

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

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

**Note:** The attribute declarations must always come last:

*<xs:attribute name="orderid" type="xs:string" use="required"/>*

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

*<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
  
<xs:element name="shiporder">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="orderperson" type="xs:string"/>  
      <xs:element name="shipto">  
        <xs:complexType>  
          <xs:sequence>  
            <xs:element name="name" type="xs:string"/>  
            <xs:element name="address" type="xs:string"/>  
            <xs:element name="city" type="xs:string"/>  
            <xs:element name="country" type="xs:string"/>  
          </xs:sequence>  
        </xs:complexType>  
      </xs:element>  
      <xs:element name="item" maxOccurs="unbounded">  
        <xs:complexType>  
          <xs:sequence>  
            <xs:element name="title" type="xs:string"/>  
            <xs:element name="note" type="xs:string" minOccurs="0"/>  
            <xs:element name="quantity" type="xs:positiveInteger"/>  
            <xs:element name="price" type="xs:decimal"/>  
          </xs:sequence>  
        </xs:complexType>  
      </xs:element>  
    </xs:sequence>  
    <xs:attribute name="orderid" type="xs:string" use="required"/>  
  </xs:complexType>  
</xs:element>  
  
</xs:schema>*

## Divide the Schema

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

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

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

*<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">*  
**<!-- definition of simple elements -->***<xs:element name="orderperson" type="xs:string"/>  
<xs:element name="name" type="xs:string"/>  
<xs:element name="address" type="xs:string"/>  
<xs:element name="city" type="xs:string"/>  
<xs:element name="country" type="xs:string"/>  
<xs:element name="title" type="xs:string"/>  
<xs:element name="note" type="xs:string"/>  
<xs:element name="quantity" type="xs:positiveInteger"/>  
<xs:element name="price" type="xs:decimal"/>*  
**<!-- definition of attributes -->***<xs:attribute name="orderid" type="xs:string"/>*  
**<!-- definition of complex elements -->***<xs:element name="shipto">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element ref="name"/>  
      <xs:element ref="address"/>  
      <xs:element ref="city"/>  
      <xs:element ref="country"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>*  
*<xs:element name="item">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element ref="title"/>  
      <xs:element ref="note" minOccurs="0"/>  
      <xs:element ref="quantity"/>  
      <xs:element ref="price"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>  
  
<xs:element name="shiporder">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element ref="orderperson"/>  
      <xs:element ref="shipto"/>  
      <xs:element ref="item" maxOccurs="unbounded"/>  
    </xs:sequence>  
    <xs:attribute ref="orderid" use="required"/>  
  </xs:complexType>  
</xs:element>  
  
</xs:schema>*

## Using Named Types

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

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

*<?xml version="1.0" encoding="UTF-8" ?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
  
<xs:simpleType name="stringtype">  
  <xs:restriction base="xs:string"/>  
</xs:simpleType>  
  
<xs:simpleType name="inttype">  
  <xs:restriction base="xs:positiveInteger"/>  
</xs:simpleType>  
  
<xs:simpleType name="dectype">  
  <xs:restriction base="xs:decimal"/>  
</xs:simpleType>  
  
<xs:simpleType name="orderidtype">  
  <xs:restriction base="xs:string">  
    <xs:pattern value="[0-9]{6}"/>  
  </xs:restriction>  
</xs:simpleType>  
  
<xs:complexType name="shiptotype">  
  <xs:sequence>  
    <xs:element name="name" type="stringtype"/>  
    <xs:element name="address" type="stringtype"/>  
    <xs:element name="city" type="stringtype"/>  
    <xs:element name="country" type="stringtype"/>  
  </xs:sequence>  
</xs:complexType>  
  
<xs:complexType name="itemtype">  
  <xs:sequence>  
    <xs:element name="title" type="stringtype"/>  
    <xs:element name="note" type="stringtype" minOccurs="0"/>  
    <xs:element name="quantity" type="inttype"/>  
    <xs:element name="price" type="dectype"/>  
  </xs:sequence>  
</xs:complexType>  
  
<xs:complexType name="shipordertype">  
  <xs:sequence>  
    <xs:element name="orderperson" type="stringtype"/>  
    <xs:element name="shipto" type="shiptotype"/>  
    <xs:element name="item" maxOccurs="unbounded" type="itemtype"/>  
  </xs:sequence>  
  <xs:attribute name="orderid" type="orderidtype" use="required"/>  
</xs:complexType>  
  
<xs:element name="shiporder" type="shipordertype"/>  
  
</xs:schema>*

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

*<xs:restriction base="xs:string">*

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

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

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

**XSD – Validation**

Java based XSD validator to validate **students.xml** against the **students.xsd**.

**students.xml**

*<?xml version = "1.0"?>*

*<class>*

*<student rollno = "393">*

*<firstname>Dinkar</firstname>*

*<lastname>Kad</lastname>*

*<nickname>Dinkar</nickname>*

*<marks>85</marks>*

*</student>*

*<student rollno = "493">*

*<firstname>Vaneet</firstname>*

*<lastname>Gupta</lastname>*

*<nickname>Vinni</nickname>*

*<marks>95</marks>*

*</student>*

*<student rollno = "593">*

*<firstname>Jasvir</firstname>*

*<lastname>Singh</lastname>*

*<nickname>Jazz</nickname>*

*<marks>90</marks>*

*</student>*

*</class>*

**students.xsd**

*<?xml version = "1.0"?>*

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

*<xs:element name = 'class'>*

*<xs:complexType>*

*<xs:sequence>*

*<xs:element name = 'student' type = 'StudentType' minOccurs = '0'*

*maxOccurs = 'unbounded' />*

*</xs:sequence>*

*</xs:complexType>*

*</xs:element>*

*<xs:complexType name = "StudentType">*

*<xs:sequence>*

*<xs:element name = "firstname" type = "xs:string"/>*

*<xs:element name = "lastname" type = "xs:string"/>*

*<xs:element name = "nickname" type = "xs:string"/>*

*<xs:element name = "marks" type = "xs:positiveInteger"/>*

*</xs:sequence>*

*<xs:attribute name = 'rollno' type = 'xs:positiveInteger'/>*

*</xs:complexType>*

*</xs:schema>*

**XSDValidator.java**

*import java.io.File;*

*import java.io.IOException;*

*import javax.xml.XMLConstants;*

*import javax.xml.transform.stream.StreamSource;*

*import javax.xml.validation.Schema;*

*import javax.xml.validation.SchemaFactory;*

*import javax.xml.validation.Validator;*

*import org.xml.sax.SAXException;*

*public class XSDValidator {*

*public static void main(String[] args) {*

*if(args.length !=2){*

*System.out.println("Usage : XSDValidator <file-name.xsd> <file-name.xml>" );*

*} else {*

*boolean isValid = validateXMLSchema(args[0],args[1]);*

*if(isValid){*

*System.out.println(args[1] + " is valid against " + args[0]);*

*} else {*

*System.out.println(args[1] + " is not valid against " + args[0]);*

*}*

*}*

*}*

*public static boolean validateXMLSchema(String xsdPath, String xmlPath){*

*try {*

*SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C\_XML\_SCHEMA\_NS\_URI);*

*Schema schema = factory.newSchema(new File(xsdPath));*

*Validator validator = schema.newValidator();*

*validator.validate(new StreamSource(new File(xmlPath)));*

*} catch (IOException e){*

*System.out.println("Exception: "+e.getMessage());*

*return false;*

*}catch(SAXException e1){*

*System.out.println("SAX Exception: "+e1.getMessage());*

*return false;*

*}*

*return true;*

*}*

*}*

**Steps to validate XML against XSD**

1. Copy the **XSDValidator.java** file to any location, say **E:** > java
2. Copy the **students.xml** to same location **E:** > **java**
3. Copy the **students.xsd** to same location **E:** > **java**
4. Compile **XSDValidator.java** using console. Make sure you have JDK 1.5 onwards installed on your machine and classpaths are configured. For details on how to use JAVA, see JAVA Tutorial

**E:\java\javac XSDValidator.java**

* Execute **XSDValidator** with **students.xsd** and **students.xml** passed as argument.

**E:\java\java XSDValidator students.xsd students.xml**

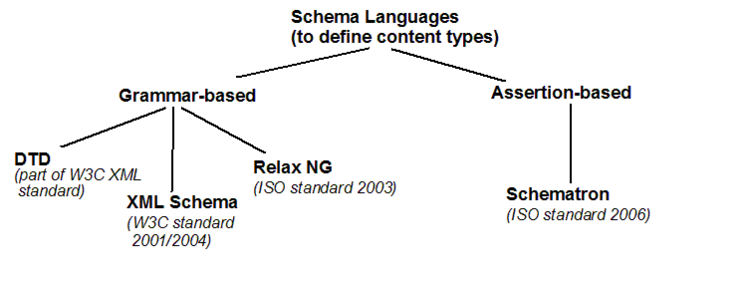
**Verify the output**

You'll see the following result −

**students.xml is valid against students.xsd**

### Types of XML grammars

We may distinguish between several kinds of XML grammars

[](http://edutechwiki.unige.ch/en/File:Xml-schema-2.png)

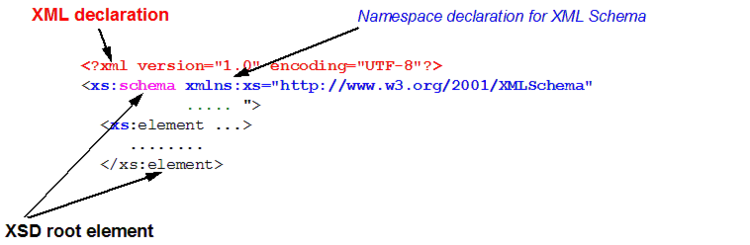
**Kinds of XML grammars**

* A grammar-based schema specifies **what elements** may be used in an XML document, the **order** of the elements, the number of **occurrences** of each element, and finally the **content and datatype** of each element and attribute.
* An assertion-based schema makes assertions about the relationships that must hold between the elements and attributes in an XML instance document.

**Comparison between grammar-based schemas**

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **DTD** | **XML Schema** | **Relax NG** |
| Adoption | wide spread | Data-centric applications like web services | R&D mostly |
| Complexity of structure | Medium | Powerful (e.g. sets, element occurrence constraints) | Powerful |
| Data types | Little (10, mostly attribute values) | Powerful (44 + your own derived data types) | Powerful (same as XSD) |
| Overall complexity | low | high | medium |
| XML-based formalism | no | yes | yes  (also a short notation) |
| Association with XML document | DOCTYPE declaration | Namespace declaration | No standard solution |
| Browser support | IE (not Firefox) | no | no |
| File suffix | \*.dtd | \*.xsd | \*.rng / \*.rnc |
| Entities | yes | no (use xinclude instead) | no |

* XML Schemas were created to define more precise grammars than with DTDs, in particular one can define Data Types and more sophisticated element structures
* DTD supports 10 datatypes, mostly for attributes. XML Schema supports 44 datatypes and in addition, you can define your own.
* Relax NG was a reaction by people who didn't like this new format. It is about as powerful as XSD. but not as complicated.

[](http://edutechwiki.unige.ch/en/File:Xml-schema-3.png)

**Note:**

**Element children can be defined in two ways:**

1. The Russian puppet model: use a '**complexType'** child element
2. The Salami model: use a '**type'** attribute that refers to a data type you defined as '**complexType'**.

Let's examine both ways:

**The Russan puppet model: <xs:complexType> (1)**

*complexType* is a child element of *element* and it will define the possible "data structures" for the element. In the example below we define five child elements for "recipe" - i.e. recipe\_name, ingredietns and directions. We also could define attributes, grand children that way.

<xs:element name="recipe">

<xs:complexType>

<xs:sequence>

<xs:element ref="meta"/>

<xs:element minOccurs="0" ref="recipe\_author"/>

<xs:element ref="recipe\_name"/>

<xs:element ref="ingredients"/>

<xs:element ref="directions"/>

</xs:sequence>

</xs:complexType>

</xs:element>

The Russian puppet model is recommended for very simple DTDs. The Salami model below is more modular and therefore a better solution, most of the time.

**The Salami model: <xs:complexType> (2)**

You can declare a complex type by itself and then "use it" in an element declaration.

Example XSD: [recipe2.xsd](http://tecfa.unige.ch/guides/xml/examples/xsd-examples/recipe2.xsd)

* Defining an element that refers to a complex type for its child elements:

<xs:element name="recipe" type="recipe\_contents" />

* Defining the complex type:

<xs:complexType name="recipe\_contents">

<xs:sequence>

<xs:element ref="meta"/>

<xs:element minOccurs="0" ref="recipe\_author"/>

<xs:element ref="recipe\_name"/>

<xs:element ref="meal"/>

<xs:element ref="ingredients"/>

<xs:element ref="directions"/>

</xs:sequence>

</xs:complexType>

## Converting DTDs to XSDs

Most decent XML editors have a built-in translator that will do most of the work. However, generated XSD code is not necessarily the most pretty ...

* e.g. in Exchanger XML Editor: Use Menu Schema -> Convert Schema. The result is fairly good. Make sure to validate the DTD, before you translate !

Below we present a table including XSD definitions for typical DTD structural elements. In the examples we use a namespace prefix for the XML and none for the Schema. Therefore an \*.xsd file would look like this:

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://www.w3.org/2001/XMLSchema file:/usr/local/xngr/types/XML%20Schema/Validation/XMLSchema.xsd"*

*xmlns:t="http://testing.org/"*

*targetNamespace="http://testing.org/" >*

*<element name="ROOT">*

*<complexType>*

*<sequence>*

*<element ref="t:A"/>*

*<element ref="t:B"/>*

*</sequence>*

*</complexType>*

*</element>*

*<element name="A" type="string"/>*

*<element name="B" type="string"/>*

*</schema>*

|  |  |
| --- | --- |
| **DTD** | **XML Schema** |
| <!ELEMENT ROOT (A,B) > | <element name="ROOT">  <complexType>  <sequence>  <element ref="t:A"/>  <element ref="t:B"/>  </sequence>  </complexType>  </element> |
| <!ELEMENT ROOT (A|B) > | <element name="ROOT">  <complexType>  <choice>  <element ref="t:A"/>  <element ref="t:B"/>  </choice>  </complexType>  </element> |
| <!ELEMENT ROOT (A|(B,C)) > | <element name="ROOT">  <complexType>  <choice>  <element ref="t:A"/>  <sequence>  <element ref="t:B"/>  <element ref="t:C"/>  </sequence>  </choice>  </complexType>  </element> |
| <!ELEMENT ROOT (A?,B+,C\*) > | <element name="ROOT">  <complexType>  <sequence>  <element ref="t:A" minOccurs="0"/>  <element ref="t:B" maxOccurs="unbounded"/>  <element ref="t:C" minOccurs="0" maxOccurs="unbounded"/>  </sequence>  </complexType>  <element> |

**Attribute definitions**

|  |  |
| --- | --- |
| **DTD** | **XML Schema** |
| <!ATTLIST ROOT a CDATA #REQUIRED> | <element name="ROOT">  <complexType content="elementOnly">  <attribute name="a" type="string" use="required"/>  </complexType>  </element> |
| <!ATTLIST ROOT a CDATA #IMPLIED> | <element name="ROOT">  <complexType content="elementOnly">  <attribute name="a" type="string" use="optional"/>  </complexType>  </element> |
| <!ATTLIST ROOT a (x|y|z)#REQUIRED;> | <element name="ROOT">  <complexType content="elementOnly">  <attribute name="a">  <simpleType base="string">  <enumeration value="x"/>  <enumeration value="y"/>  <enumeration value="z"/>  </simpleType>  </attribute>  </complexType>  </element> |
| <!ATTLIST ROOT a CDATA #FIXED "x"> | <element name="ROOT">  <complexType content="elementOnly">  <attribute name="a" type="string"  use="fixed" value="x"/>  </complexType>  </element> |

Reminder: as we explained above, either the XSD or the target language must use a namespace prefix for the elements names (not the attributes). E.g. The first rule above could also have been written like this:

|  |  |
| --- | --- |
| **DTD** | **XML Schema** |
| <!ELEMENT ROOT (A,B) > | <xs:element name="ROOT">  <xs:complexType>  <xs:sequence>  <xs:element ref="A"/>  <xs:element ref="B"/>  </xs:sequence>  </xs:complexType>  </xs:element> |

**XSD – WEB SERVICES:**

**XML Schema Document (XSD)**

XSD stands for XML schema document. XSD is also an XML document. It is owned by W3Org. The latest version of XSD is 1.1.

Even though we can use DTD’s for defining the validity of an XML document, it is not type safe and is not flexibile. XSD is more powerful and is type strict in nature.

XSD document is used for defining the structure of an XML document; it declares elements and the type of data that elements carry.

As XSD is also an XML document so, it also starts with prolog and has one and only one root element. In case of XSD document the root element is <schema>. All the subsequent elements will be declared under this root element.

**An XML contains two types of elements.**

A) Simple elements (these contain content as data)

B) Compound or Container’s (these contains sub-elements under it).

So, while writing an XSD documents we need to represent two types of elements simple or compound elements.

**The syntax’s for representing a simple and compound elements of XML in a XSD document are shown below.**

**Syntax for representing simple elements of XML document**

*<xs:element name=”elementname” type=”datatype”/>*

**Syntax for representing compound element of XML document**

In order to represent compound element of an XML document in an XSD, first we need to create a type declaration representing structure of that xml element. Then

we need to declare element of that user-defined type, shown below.

*<xs:complexType name=”typeName”>*

*<xs:sequence> or <xs:all>*

*<xs:element name=”..” type=”..”/>*

*<xs:element name=”..” type=”..”/>*

*<xs:element name=”..” type=”..”/>*

*</xs:sequence> or </xs:all>*

*</xs:complexType>*

Declaring a complex type is equal to declaring a class in java.

In java we define user-defined data types using class declaration. When we declare a class it represents the structure of our data type, but it doesn’t allocates any memory.

Once a class has been declared you can create any number of objects of that class.

The same can be applied in case of complex type declaration in an XSD document.

In order to create user-defined data type in XSD document you need to declare a complex type, creating a complex indicates you just declared the class structure.

Once you create your own data type, now you can define as many elements you need of that type.

**Let’s say for example I declared a complex type whose name is AddressType as follows**

*<xs:complexType name=”AddressType”>*

*<xs:sequence>*

*<xs:element name=”addressLine1” type=”xs:string”/>*

*<xs:element name=”addressLine2” type=”xs:string”/>*

*<xs:element name=”city” type=”xs:string”/>*

*</xs:sequence>*

*</xs:complexType>*

Now the above fragment indicates you created your own user-defined data type whose name is “AddressType”. Now you can declare any number of elements representing that type as shown below.

*<xs:element name=”myAddress” type=”AddressType”/>*

**Let’s take an XML document for which we will show how its XSD looks like**

**XML Document – Courier consignment information**

*<consignment>*

*<id>C4242</id>*

*<bookedby>durga</bookedby>*

*<deliveredTo>Sriman</deliveredTo>*

*<shippingAddress>*

*<addressLine1>S.R Nagar</addressLine1>*

*<addressLine2>Opp Chaitanya</addressLine2>*

*<city>Hyderabad</city>*

*<state>Andhra Pradesh</state>*

*<zip>353</zip>*

*<country>India</country>*

*</shippingAddress>*

*</consignment>*

**XSD Document – Courier consignment information**

*<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”>*

*<xs:element name=”consignment” type=”consignmentType”/>*

*<xs:complexType name=”consignmentType”>*

*<xs:sequence>*

*<xs:element name=”id” type=”xs:string”/>*

*<xs:element name=”bookedby” type=”xs:string”/>*

*<xs:element name=”deliveredTo” type=”xs:string”/>*

*<xs:element name=”shippingAddress”*

*type=”shippingAddressType”/>*

*</xs:sequence>*

*</xs:complexType>*

*<xs:complexType name=”shippingAddressType”>*

*<xs:sequence>*

*<xs:element name=”addressLine1” type=”xs:string”>*

*<xs:element name=”addressLine2” type=”xs:string”>*

*<xs:element name=”city” type=”xs:string”>*

*<xs:element name=”state” type=”xs:string”>*

*<xs:element name=”zip” type=”xs:int”>*

*<xs:element name=”country” type=”xs:string”>*

*</xs:sequence>*

*</xs:complexType>*

*</xs:schema>*

**Sequence VS All**

When declaring a complex type, following the <xs:complexType> tag we see a tag

<xs:sequence> or <xs:all>. When we use <xs:sequence> under <xs:complexType> tag, what it indicates is all the elements that are declared in that complex type must appear in the same order in xml document.

For example let’s take a complex type declaration”ItemType” which uses <xs:sequence> tag in its declaration.

*<xs:element name=”item” type=”ItemType”/>*

*<xs:complexType name=”ItemType”>*

*<xs:sequence>*

*<xs:element name=”itemCode” type=”xs:string”/>*

*<xs:element name=”quantity” type=”xs:int”/>*

*</xs:sequence>*

*</xs:complexType>*

So while using the item element in the xml we need to declare the itemCode and quantity sub-elements under item in the same order as shown below.

*<item>*

*<itemCode>IC303</itemCode>*

*<quantity>35</quantity>*

*</item>*

When we use <xs:all>, under item element the itemCode and quantity may not

appear in the same order. First quantity might appear then itemCode can present

in the xml.

**Extending Complex Types**

XSD along with allowing you to declare your own data types, it will also allow you to extend your own types by the means of inheritance as shown below.

*<xs:complexType name=”USShippingAddressType”>*

*<xs:complexContent>*

*<xs:extension base=”shippingAddressType”>*

*<xs:sequence>*

*<xs:element name=”county” type=”xs:int”/>*

*</xs:sequence>*

*</xs:extension>*

*</xs:complexContent>*

*</xs:complexType>*

**Imposing restrictions on SimpleTypes**

XSD’s allows you to define data validations on the data an element is going to carry. In order to do this you can define your own simpleType by extending the inbuilt types and can impose restrictions on it as shown below.

*<xs:simpleType name=”zipType”>*

*<xs:restriction base=”xs:int”>*

*<xs:totalDigits value=”4”/>*

*</xs:restriction>*

*</xs:simpleType>*

**XSD Namespace**

Every programming language one or in another way allows you to declare userdefined data types. Let’s consider the case of “C” language it allows you to declare your own types using Structure. In case of “C++” Class declaration allows you to declare your own type, same in case of Java as well.

When it comes to XSD you can declare your element type using XSD complex type declaration.

As how any language allows you to create your own types, they allow you to resolve the naming collision between their types. Let’s consider the case of Java; it allows you to declare your own types by class declarations. But when a programmer is given a choice of declaring their own types, language has to provide a means to resolve type names collision declared by several programmers. Java allows resolving those type naming conflicts using packages.

Packages are the means of grouping together the related types. It will allow you to uniquely identify a type by prefixing the package name. In the same way XSD also allows you to declare your own data type by using Complex Type declaration and in the same way it allows you to resolve the type naming conflicts by means of Namespace declaration.

**XSD Namespaces has two faces,**

1) Declaring the namespace in the XSD document using Targetnamespace declaration.

2) Using the elements that are declared under a namespace in xml document.

**XSD Targetnamespace**

Targetnamespace declaration is similar to a package declaration in java. You will bind your classes to a package so, that while using them you will refer with fully qualified name. Similarly when you create a complexType or an Element you will bind them to a namespace, so that while referring them you need to use qName.

In order to declare a package we use package keyword followed by name of the package. To declare a namespace in XSD we need to use targetNameSpace attribute at the Schema level followed by targetnamespace label as shown below.

***Example:-***

*<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”*

***targetnamespace****=”http://durgasoft.com/training/calendar/types”>*

*<xs:complexType name=”courseType”>*

*<xs:element name=”courseId” type=”xs:string”/>*

*<xs:element name=”courseName” type=”xs:string”/>*

*<xs:element name=”duration” type=”xs:datetime”/>*

*</xs:complexType>*

*</xs:schema>*

The courseType by default is binded to the namespace http://durgasoft.com/training/calendar/types

So, while creating an element “course” of type courseType you should use the qName of the courseType rather simple name.

***(qName means namespace:element/type name).***

But the namespace labels could be any of the characters in length, so if we prefix the entire namespace label to element/type names it would become tough to read.

So, to avoid this problem XSD has introduced a concept called short name. Instead of referring to namespace labels you can define a short name for that namespace label using xmlns declaration at the <schema> level and you can use the short name as prefix instead of the complete namespace label as shown below.

*<xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”*

***targetnamespace****=http://durgasoft.com/training/calendar/types*

***xmlns:dt=”http://durgasoft.com/training/calendar/types”****>*

***<xs:element name=”course” type=”dt:courseType”/>***

*<xs:complexType name=”courseType”>*

*<xs:element name=”courseId” type=”xs:string”/>*

*<xs:element name=”courseName” type=”xs:string”/>*

*<xs:element name=”duration” type=”xs:datetime”/>*

*</xs:complexType>*

*</xs:schema>*

In java we can have only one package declaration at a class level. In the same way we can have only one targetNamespace declaration at an XSD document.

**Using elements from an xml namespace (xmlns)**

While writing an XML document, you need to link it to XSD to indicate it is following the structure defined in that XSD. **In order to link an XML to an XSD we use an attribute “schemaLocation”.**

**schemaLocation attribute declaration has two pieces of information.**

First part is representing the namespace from which you are using the elements. Second is the document which contains this namespace, shown below.

*<?xml version=”1.0” encoded=”utf-8”?>*

*<course xsi:schemaLocation=”http://durgasoft.com/training/calendar/types*

*file:///c:\folder1\folder2\courseInfo.xsd”*

*xmlns:xsi=”http://www.w3.org/2001/XMLSchema-Instance”>*

*</course>*

In the above xml we stated this xml is using xsd document courseInfo.xsd whose namespace is http://durgasoft.com/training/calendar/types

If you want to include two XSD documents in the xml then you need to declare in schemaLocation tag <namespace1> <schemalocation1> <namespace2>

<schemalocation2>.

***For example:-***

*<?xml version=”1.0” encoded=”utf-8”?>*

*<course xsi:schemaLocation=”http://durgasoft.com/training/calendar/types*

*file:///c:\folder1\folder2\courseInfo.xsd*

*http://durgasoft.com/training/vacation/types*

*file:///c:\folder1\folder2\vacation.xsd”*

*xmlns:xsi=”http://www.w3.org/2001/XMLSchema-Instance”>*

*</course>*

Now in the above xml we are using two XSD documents courseInfo.xsd and vacation.xsd. With this declaration we will not be able to find whether the course element is used from courseInfo.xsd or vacation.xsd. To indicate it we should prefix the namespace to the course element.

But the namespace labels can be arbitrary string of characters in any length. So, we need to define short name, so that we can prefix shortname while referring the elements as shown below.

*<?xml version=”1.0” encoded=”utf-8”?>*

*<dc:course xsi:schemaLocation=”http://durgasoft.com/training/calendar/types*

*file:///c:\folder1\folder2\courseInfo.xsd*

*http://durgasoft.com/training/vacation/types*

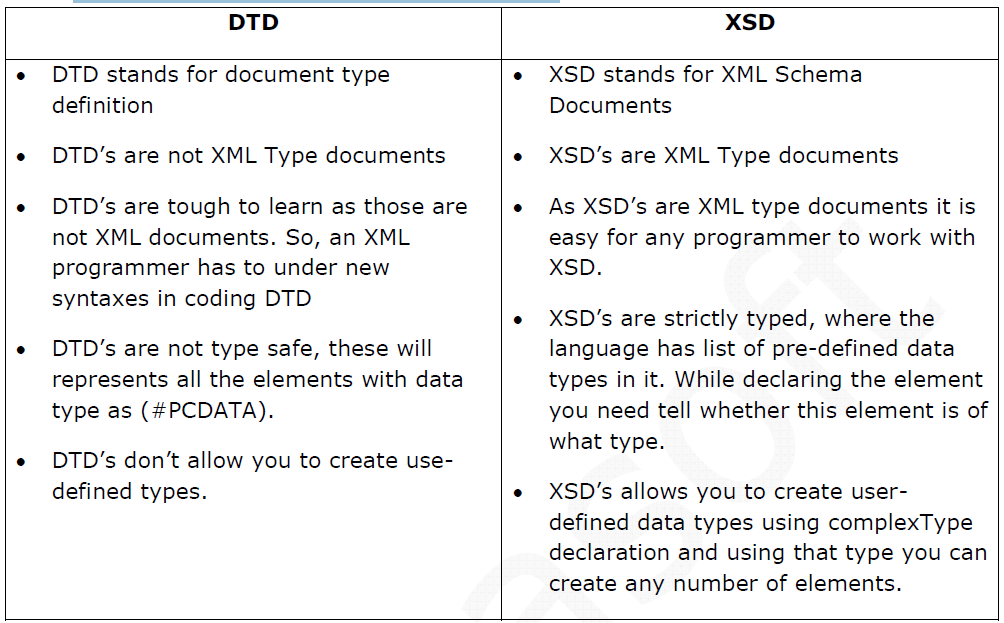
*file:///c:\folder1\folder2\vacation.xsd”*

*xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance*

***xmlns:dc=”http://durgasoft.com/training/calendar/types”****>*

*</dc:course>*

**Difference between DTD and XSD**



**XML Schema Basics:**

The XML specification includes the Document Type Definition (DTD), which can be used to describe XML markup languages and to validate instances of them (XML documents). While DTDs have proven very useful over the years, they are also limited. To address limitations of DTDs, the W3C (World Wide Web Consortium), which manages the fundamental XML standards, created a new way to describe markup languages called XML schema.

**Why XML Schema Is Preferred to DTDs in Web Services**

DTDs have done an adequate job of telling us how elements and attributes are organized in a markup language, but they fail to address data typing. For example, the describes the valid organization of the Address Markup Language we created earlier. The DTD declares that an address element may contain one or more street elements and must contain exactly one of each of the city, state, and zip elements. It also declares that the address element must have a category attribute.

**A DTD**

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

*<!ELEMENT address (street+, city, state, zip)>*

*<!ELEMENT street (#PCDATA) >*

*<!ELEMENT city (#PCDATA) >*

*<!ELEMENT state (#PCDATA) >*

*<!ELEMENT zip (#PCDATA) >*

*<!ATTLIST address category CDATA #REQUIRED >*

A parser reading an XML instance determines whether it's valid by comparing it to its DTD—if it declares that it uses a DTD. To be valid, an XML instance must conform to its DTD, which means it must use the elements specified by the DTD in the correct order and multiplicity (zero, one, or many times).

While constraints provided by DTDs are useful for validating XML instances, the probability that an XML instance will have a valid organization but contain invalid data is pretty high. DTDs have a very weak typing system that restricts elements to four broad types of data: EMPTY, ANY, element content or mixed element-and-text content. In other words, DTDs can only restrict elements to containing nothing, other elements, or text—not a very granular typing system. DTDs don't support types like integer, decimal, Boolean, and enumeration.

For example, the Address Markup DTD cannot restrict the contents of the zip element to an integer value or the state element to a set of valid state codes.

XML schema, by contrast, provides a much stronger type system. Many believe that XML schema is superior to DTD because it defines a richer type system, which includes simple primitives (integer, double, Boolean, among others) as well as facilities for more complex types. XML schema facilitates type inheritance, which allows simple or complex types to be extended or restricted to create new types.

In addition, XML schema supports the use of XML namespaces to create compound documents composed of multiple markup languages.

**The XML Schema Document**

A schema describes an XML markup language. Specifically it defines which elements and attributes are used in a markup language, how they are ordered and nested, and what their data types are.

A schema describes the structure of an XML document in terms of complex types and simple types.

Complex types describe how elements are organized and nested.

Simple types are the primitive data types contained by elements and attributes.

For example, shows a portion of a schema that describes the Durga-jobs Markup Language.

Durga-jobs Markup defines a set of XML schema types used by Durga-jobs Books: USAddress, PurchaseOrder, Invoice, Shipping, and the like.

At this point all the different types used by Durga-jobs Books are combined into one schema; later you'll learn how to separate them into their own schemas and independent markup languages.

**The Address Definition in a Schema**

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

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook">*

*<element name="address" type="mh:USAddress" />*

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*</complexType>*

*...*

*</schema>*

The first thing you may have noticed is that is actually an XML document. That schemas are XML documents is a critical point: It makes the development of validating parsers and other software tools easier, because the operations that manipulate schemas can be based on XML parsers, which are already widely available.

DTDs, the predecessor to schemas, were not based on XML, so processing them required special parsing. The root element of a schema document is always the schema element. Nested within the schema element are element and type declarations. Above example declares a complex type named USAddress, and an element of that type named address.

The schema element assigns the XML schema namespace (***"http://www.w3.org/2001/XMLSchema"***) as the default namespace. This namespace is the standard namespace defined by the XML schema specification—all the XML schema elements must belong to this namespace.

The schema element also defines the targetNamespace attribute, which declares the XML namespace of all new types explicitly created within the schema.

For example, the USAddress type is automatically assigned to targetNamespace, "http://www.Durga-jobs.com/jwsbook".

The schema element also uses an XML namespace declaration to assign the prefix mh to the targetNamespace. Subsequently, newly created types in the schema can be referred to as "mh:Typename".

For example, the type attribute in the element declaration in Listing 3-2 refers to the USAddress as "mh:USAddress":

*<element name="address" type="mh:USAddress" />*

An instance document based on this schema would use the address element directly or refer to the USAddress type. When a parser that supports XML schema reads the document, it can validate the contents of the XML document against the USAddress type shows a conforming XML instance.

**An Instance of the Address Markup Language**

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

*<addr:address xmlns:addr="http://www.Durga-jobs.com/jwsbook"> <name>Amazon.com</name>*

*<street>1516 2nd Ave</street>*

*<city>Seattle</city>*

*<state>WA</state> <zip>90952</zip>*

*</addr:address>*

Using XML schema, we can state exactly how an instance of the address element should be organized and the types of data its elements and attributes should contain.

**Simple Types**

A simple type resembles a Java primitive type in that both are atomic; they cannot be broken down into constituent parts. In other words, a simple element type will not contain other elements; it will contain only data. The XML schema specification defines many standard simple types, called built-in types. The built-in types are the standard building blocks of an XML schema document. They are members of the XML schema namespace, *"http://www.w3.org/2001/XMLSchema".*

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

*<schema xmlns=*[*http://www.w3.org/2001/XMLSchema*](http://www.w3.org/2001/XMLSchema) *xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook">*

*...*

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string" />*

*<element name="accountNumber" type="integer" />*

*<element name="total" type="float" />*

*<!-- More stuff follows -->*

*</sequence>*

*</complexType>*

*...*

*</schema>*

*package com.monsonhaefel.jwsbook;*

*public class PurchaseOrder {*

*String accountName;*

*int accountNumber;*

*float total;*

*// more stuff follows*

*}*

The PurchaseOrder complex type declares three of its elements and an attribute using the XML schema built-in types: string, integer, and float.

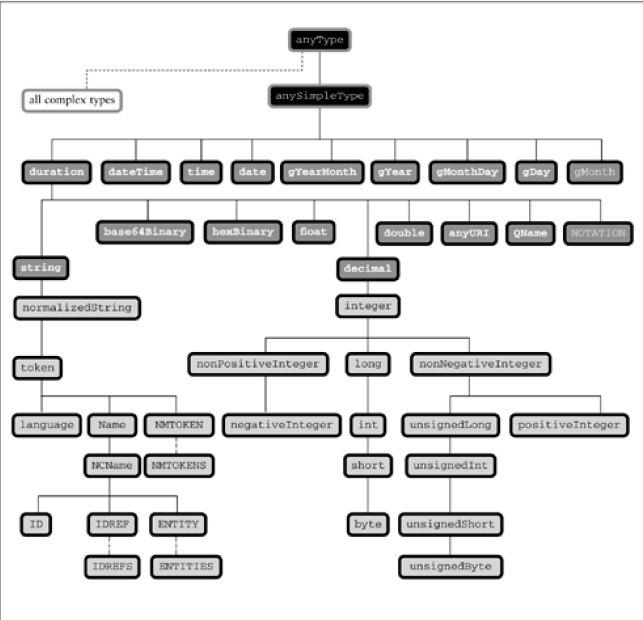
These simple types are similar to familiar types in the Java programming language and others. In a schema, simple types are used to construct complex types, much as Java primitives are used as fields of Java class definitions.

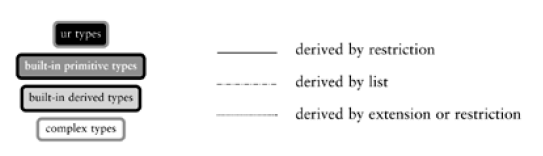
|  |  |
| --- | --- |
| The XML schema specification describes its 44 built-in simple types in precise detail. This precision enables XML parsers to process the built-in types predictably and consistently, for the most part, and provides a solid foundation for creating your own complex and custom simple types.  For example, the XML schema specification tells us that a string is defined as an unlimited length of characters based on the Universal Character Set;  an unsignedShort is a non-decimal number between 0 and 65,535;  a float is a 32-bit floating-point type; and  a date is represented as YYYY-MM-DD.  The Universal Character Set (ISO/IEC 10646-1993) is a superset of all other character codes, including UTF-8 and UTF-16.  **Comparing the Use of XML Schema Simple Types and Java Primitive Types**  XML Schema Built-in Simple Types (shown in bold) Java Primitive Types (shown in bold) A Subset of the XML Schema Built-in Simple Types | |
| **Simple Type** | **Definition** |
| ***string*** | A sequence of characters conforming to UCS |
| ***normalizedString*** | A string without carriage returns, line feeds, or tabs |
| ***token*** | A string without spaces, line feeds, or tabs |
| ***NMTOKEN*** | A token used in attributes |
| ***byte*** | A non-decimal number between –128 and 127 |
| ***unsignedByte*** | A non-decimal number between 0 and 255 |
| ***base64Binary*** | Base64-encoded binary data (RFC 2045)[a] |
| ***hexBinary*** | Hex-encoded binary data[b] |
| ***integer*** | A base-10-integer number of any size (…)[c] |
| ***positiveInteger*** | A base-10 integer greater then zero (1, 2, …) |
| ***negativeInteger*** | A base-10 integer less then zero (…, –2, –1) |

**A Subset of the XML Schema Built-in Simple Types**

|  |  |  |
| --- | --- | --- |
| **Simple Type** | **Definition** | |
| **int** | A base-10 integer between –2,147,483,648 and 2,147,483,647 (–2 billion and 2 billion) | |
| **unsignedInt** | A base-10 integer between 0 and 4,294,967,295 (zero and 4 billion) | |
| **long** | A base-10 integer between –9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (–9 quintillion and 9 quintillion) | |
| **unsignedLong** | A base-10 integer between 0 and 18,446,744,073,709,551,615 (zero and 18 quintillion) | |
| **short** | A base-10 integer between –32,767 and 32,767 | |
| **unsignedShort** | A base-10 integer between 0 and 65,535 | |
| **decimal** | A decimal number of any precision and size | |
| **float** | A decimal number conforming to the IEEE single-precision 32-bit floating-point type[d] | |
| **double** | A decimal number conforming to the IEEE double-precision 64-bit floating-point type[d] | |
| **boolean** | A boolean value of "true" or "false" | |
| You can also use the values of "0" (false) or "1" (true); either convention is fine. | | | |
| **time** | | A time in hours, minutes, seconds, and milliseconds formatted as hh:mm:ss.sss (e.g., 1:20 PM is 13:20:00) | |
| You may include the optional Coordinated Universal Time (UTC) designator (e.g., 1:20 PM Eastern Standard Time (EST) is 13:20:00-05:00)[e] | | | |
| **date** | | A Gregorian date in centuries, years, months, and days (e.g., December 31, 2004 is 2004-12-31)[e] | |
| **dateTime** | | A Gregorian date measured in centuries, years, months, and days, with a time field set off by a T (e.g., 1:20 PM EST on December 31, 2004 would be 2004-12-31T13:20:00-05:00)[e] | |
| **duration** | | A span of time measured in years, months, days, and seconds (e.g., 1 year, 2 months, 3 days, 10 hours, and 30 minutes would be P1Y2M3DT10H30M) | |
| Duration may be negative, and zero values can be left off (e.g., 120 days earlier is P120D). The value must always start with the letter P.[f] | | | |

**XML Schema Type Hierarchy**





**Complex Types**

A schema may declare complex types, which define how elements that contain other elements are organized.

The USAddress schema type in Listing 3-2, for example, is a complex type definition for a United States postal address. It tells us that an element based on this type will contain five other elements called name, street, city, state, and zip.

A complex type is analogous to a Java class definition with fields but no methods. The fields in a Java class declare the names and types of variables that an instance of that class will contain.

Similarly, a complex type declares the names and types of elements and attributes that an XML instance of that type may contain. An instance of a complex type is an element in an XML document.

**XML Schema: Complex Type Java Class Definition**

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*</complexType>*

*public class USAddress {*

*public String name;*

*public String street;*

*public String city;*

*public String state;*

*public String zip;*

*}*

While this analogy between XML schema complex types and Java class definitions is helpful, take care not to confuse them. A schema is used to define elements and attributes in a markup language and verify the correctness of an XML instance; it's not a computer program.

**Sequences of Elements**

Most complexType declarations in schemas will contain a sequence element that lists one or more element definitions. The element definitions tell you which elements are nested in the type, the order in which they appear, and the kind of data each element contains.

The USAddress type clearly defines the proper structure of a U.S. postal address and can be used to verify the proper contents of any element based on that type. For example, the address element used throughout Chapter 2 could be an instance of the type USAddress, and we could use that type to verify the contents of the address element when it was used in an XML instance.

A complex type may contain a sequence of elements that are simple types or other complex types.

For example, we can define an element for a purchase-order document by adding a PurchaseOrder type to the Durga-jobs Markup Language you saw in Listing 3-2. In Listing 3-4, the new PurchaseOrder type has two nested elements, billAddress and shipAddress, both of type USAddress. Listing 3-4.

**The PurchaseOrder Type in a Schema Code View: Scroll / Show All**

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

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook" >*

*<element name="purchaseOrder" type="mh:PurchaseOrder" />*

*<element name="address" type="mh:USAddress" />*

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string" />*

*<element name="accountNumber" type="unsignedShort" />*

*<element name="shipAddress" type="mh:USAddress" />*

*<element name="billAddress" type="mh:USAddress" />*

*<element name="book" type="mh:Book" />*

*<element name="total" type="float" />*

*</sequence>*

*</complexType>*

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*</complexType>*

**Mapping a Schema Complex Type to an XML Element XML Schema: USAddress XML Document: address**

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*</complexType>*

*<address>*

*<name>Amazon.com</name>*

*<street>1516 2nd Ave</street>*

*<city>Seattle</city>*

*<state>WA</state>*

*<zip>90952</zip>*

*</address>*

*<complexType name="Book">*

*<sequence>*

*<element name="title" type="string" />*

*<element name="quantity" type="unsignedShort" />*

*<element name="wholesale-price" type="float" />*

*</sequence>*

*</complexType>*

*</schema>*

The schema makes use of both complex types (PurchaseOrder, USAddress, and Book) and simple types (string, unsignedShort, and float). The USAddress type is a member of the targetNamespace, so we refer to it by its fully qualified name, "mh:USAddress". (Recall that targetNamespace is assigned the namespace prefix mh in the schema element.)

As you can see, the PurchaseOrder type takes full advantage of USAddress by using it to define both its billAddress and shipAddress elements. In this way, complex type declarations can build on other complex type definitions to create rich types that easily describe very complex XML structures.

The PurchaseOrder type also uses Book, another complex type that describes the book being ordered. The names of XML schema types are case-sensitive. When an element declares that it is of a particular type, it must specify both the namespace and the name of that type exactly as the type declares them.

**Attributes**

In addition to sequences of elements, a complex type may also define its own attributes.

**Adding an Attribute to a Complex Type Code View: Scroll / Show All**

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

*<schema xmlns=*[*http://www.w3.org/2001/XMLSchema*](http://www.w3.org/2001/XMLSchema) *xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook">*

*<element name="purchaseOrder" type="mh:PurchaseOrder"/>*

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string"/>*

*<element name="accountNumber" type="unsignedShort"/>*

*<element name="shipAddress" type="mh:USAddress"/>*

*<element name="billAddress" type="mh:USAddress"/>*

*<element name="book" type="mh:Book"/>*

*<element name="total" type="float"/>*

*</sequence>*

*<attribute name="orderDate" type="date"/>*

*</complexType>*

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string"/>*

*<element name="street" type="string"/>*

*<element name="city" type="string"/>*

*<element name="state" type="string"/>*

*<element name="zip" type="string"/>*

*</sequence>*

*</complexType>*

*<complexType name="Book">*

*<sequence>*

*<element name="title" type="string"/>*

*<element name="quantity" type="unsignedShort"/>*

*<element name="wholesale-price" type="float"/>*

*</sequence>*

*</complexType>*

*</schema>*

The next code sample shows a valid XML document based on the PurchaseOrder type defined by the schema. The XML document would contain all the elements and the orderDate attribute as described by the PurchaseOrder complex type—and would be verifiable against that type.

**An Instance of the Schema Code View: Scroll / Show All**

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

*<po:purchaseOrder orderDate="2003-09-22" xmlns:po="http://www.Durga-jobs.com/jwsbook">*

*<accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<shipAddress>*

*<name>AMAZON.COM</name>*

*<street>1850 Mercer Drive</street>*

*<city>Lexington</city>*

*<state>KY</state>*

*<zip>40511</zip>*

*</shipAddress>*

*<billAddress>*

*<name>Amazon.com</name>*

*<street>1516 2nd Ave</street>*

*<city>Seattle</city>*

*<state>WA</state>*

*<zip>90952</zip>*

*</billAddress>*

*<book>*

*<title>J2EE Web Services</title>*

*<quantity>300</quantity>*

*<wholesale-price>24.99</wholesale-price>*

*</book>*

*<total>8997.00</total>*

*</po:purchaseOrder>*

**Occurrence Constraints**

The multiplicity of an element, the number of times it occurs in an instance document, is controlled by occurrence constraints, which are declared by the maxOccurs and minOccurs attributes.

For example, we can enhance the USAddress complex type by placing occurrence constraints on the street element as shown below.

**Using Occurrence Constraints**

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" minOccurs="1" maxOccurs="2" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*</complexType>*

The occurrence constraints specify that in any instance of USAddress the street element must be present at least once and at most twice. In other words, a USAddress can contain either one or two street elements. The default value for both maxOccurs and minOccurs is "1", so if these attributes are not specified the element must be present exactly once. Thus, by default, each USAddress must have exactly one name, city, state, and zip. The minOccurs attribute may be "0", indicating that an element is optional, or any positive integer value that is less than or equal to the maxOccurs value. The maxOccurs value may be any positive integer greater than or equal to the min Occurs value. minOccurs 0 maxOccurs minOccurs You may also define a maxOccurs value to be "unbounded" to specify that the element may occur an unlimited number of times.

For example, suppose Durga-jobs Books wants to avoid storing a billing address that is identical to the shipping address, and to allow customers to buy an unlimited number of books on a single order. We can redefine the PurchaseOrder type, setting the occurrence constraints on the billAddress and the book elements as highlighted below.

**Using the "unbounded" Occurrence Value**

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string" />*

*<element name="accountNumber" type="unsignedShort" />*

*<element name="shipAddress" type="mh:USAddress" />*

*<element name="billAddress" type="mh:USAddress" minOccurs="0" />*

*<element name="book" type="mh:Book" maxOccurs="unbounded" />*

*<element name="total" type="float" />*

*</sequence>*

*<attribute name="orderDate" type="date" />*

*</complexType>*

The billAddress element is now optional. It may occur at most once, because its maxOccurs value is "1" by default, but it may also be omitted because its minOccurs value is "0". The book element must be present at least once because the default value of minOccurs is "1", but it may be repeated many times because its maxOccurs is "unbounded".

Attributes also have occurrence constraints, but they are different from those of elements. Instead of maxOccurs and minOccurs, attribute types declare the use occurrence constraint, which may be "required", "optional", or "prohibited", indicating that the attribute must, may, or may not be used, respectively. The default is "optional". An attribute might be "prohibited" if you want to stop the use of a particular attribute, perhaps one that is inappropriate or no longer in use. In PurchaseOrder we want to make the orderDate attribute mandatory, so sets its use occurrence constraint to "required".

**Declaring the use Value of an Attribute:**

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string" />*

*<element name="accountNumber" type="unsignedShort" />*

*<element name="shipAddress" type="mh:USAddress" />*

*<element name="billAddress" type="mh:USAddress" minOccurs="0" />*

*<element name="book" type="mh:Book" maxOccurs="unbounded" />*

*<element name="total" type="float" />*

*</sequence>*

*<attribute name="orderDate" type="date" use="required" />*

*</complexType>*

An attribute may also have a default value, to be assigned if no value is explicitly declared in the instance document. For example, the USAddress type may include an attribute called category that can have the value "business", "private", or "government". Almost all addresses used by Durga-jobs Books are business addresses, so we set the default for the category attribute to "business" as shown below.

**Declaring the Default Value of an Attribute**

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" />*

*<element name="state" type="string" />*

*<element name="zip" type="string" />*

*</sequence>*

*<attribute name="category" type="string" default="business" />*

*</complexType>*

The default attribute can be used only when the use attribute is "optional" (recall that "optional" is the default value for the use attribute). It wouldn't make sense to declare a default when the use is "required" or "prohibited".

If the use attribute is "required", there is no need for a default because the attribute must appear in the instance document. If the use is "prohibited", the attribute's not allowed so there is no sense having a default value.

**An attribute may also be declared fixed:**

A fixed value is assigned to the attribute no matter what value appears in the XML instance document. This feature is useful in rare situations where you want to force a particular attribute always to have the same value. For example, if a particular schema is assigned a version number, then that version number should be fixed for that schema (UDDI does this).

**The all Element**

Most of the time you'll base complex types on sequence elements, but occasionally you may want to use the all element. Unlike sequence, which defines the exact order of child elements, the XML schema all element allows the elements in it to appear in any order. Each element in an all group may occur once or not at all; no other multiplicity is allowed. In other words, minOccurs is always "0" and maxOccurs is always "1". Finally, only single elements may be used in an all group; it can't include other groupings like sequence or all, shows the schema for the address element using the all element grouping instead of sequence.

**Using the XML Schema all Element**

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

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook" >*

*...*

*<complexType name="USAddress">*

*<all>*

*<element name="name" type="string" />*

*<element name="street" type="string" />*

*<element name="city" type="string" minOccurs="0"/>*

*<element name="state" type="string" minOccurs="0"/>*

*<element name="zip" type="string" />*

*</all>*

*</complexType>*

*...*

*</schema>*

The name, street, and zip elements must be present in the instance document, but the city and state elements may be absent. The elements can be in any order, but none of the elements may occur more than once.

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

*<addr:address xmlns:addr="http://www.Durga-jobs.com/jwsbook" > <zip>90952</zip>*

*<street>1516 2nd Ave</street>*

*<name>Amazon.com</name>*

*</addr:address>*

Notice the missing city and state elements and that the order of the elements is different from that in the type definition.

**Declaring Global Elements in a Schema**

In addition to declaring simple and complex types, a schema may also declare global elements, which XML instance documents can refer to directly. Global elements are declared as direct children of the schema element, rather than children of a complex type.

For example, the following shows a portion of the schema defined in Listing 3-5, which declared the purchaseOrder element (shown in bold) to be global.

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

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook">*

*<element name="purchaseOrder" type="mh:PurchaseOrder"/>*

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string"/>*

*<element name="accountNumber" type="unsignedShort"/>*

*<element name="shipAddress" type="mh:USAddress"/>*

*<element name="billAddress" type="mh:USAddress"/>*

*<element name="book" type="mh:Book"/>*

*<element name="total" type="float"/>*

*</sequence>*

*<attribute name="orderDate" type="date"/>*

*</complexType>*

*...*

*</schema>*

An XML document based on Listing 3-5 can use the purchaseOrder element as in Listing 3-6.

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

*<po:purchaseOrder orderDate="2003-09-22" xmlns:addr="http://www.Durga-jobs.com/jwsbook">*

*<accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<shipAddress>*

*...*

*</po:purchaseOrder>*

The root element of a valid XML document must have a corresponding global element declaration in the schema. A schema may define more than one global element. For example, we can modify the schema for Durga-jobs Books so that it declares two global elements: purchaseOrder and address.

**Defining Multiple Element Declarations Code View: Scroll / Show All**

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

*<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:mh="http://www.Durga-jobs.com/jwsbook" targetNamespace="http://www.Durga-jobs.com/jwsbook">*

*<element name="address" type="mh:USAddress"/>*

*<element name="purchaseOrder" type="mh:PurchaseOrder"/>*

*<complexType name="PurchaseOrder">*

*<sequence>*

*<element name="accountName" type="string"/>*

*<element name="accountNumber" type="unsignedShort"/>*

*<element name="shipAddress" type="mh:USAddress"/>*

*<element name="billAddress" type="mh:USAddress"/>*

*<element name="book" type="mh:Book"/>*

*<element name="total" type="float"/>*

*</sequence>*

*<attribute name="orderDate" type="date"/>*

*</complexType>*

*<complexType name="USAddress">*

*<sequence>*

*<element name="name" type="string"/>*

*<element name="street" type="string"/>*

*<element name="city" type="string"/>*

*<element name="state" type="string"/>*

*<element name="zip" type="string"/>*

*</sequence>*

*</complexType>*

*...*

*</schema>*

The schema above allows you to create XML documents in which the purchaseOrder element is the root, but it also allows you to create XML documents in which the address element is the root.

**An Address Document Based on the Durga-jobs Books Schema**

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

*<addr:address xmlns:addr="http://www.Durga-jobs.com/jwsbook">*

*<name>AMAZON.COM</name>*

*<street>1850 Mercer Drive</street>*

*<city>Lexington</city>*

*<state>KY</state>*

*<zip>40511</zip>*

*</addr:address>*

By declaring two different global elements in the Durga-jobs Books schema, you effectively create two schema-verifiable markup languages, a Purchase Order Markup Language and a U.S. Address Markup Language. The implication here is that a single schema can be used to validate two—indeed many—different kinds of documents. XML schema also supports global attributes that can be referred to anywhere in the schema, and that provide a consistent attribute name and type across elements.

An example of a standard global attribute is xml:lang, which any element can use to indicate the language used in an element's value ("es" for Spanish, "en" for English, and so on). In a nutshell, global elements and attributes are declared as direct children of the schema element, while local elements and attributes are not; they are the children of complex types.

**Assigning and Locating Schemas**

The whole point of schemas is that they define the grammar by which XML documents can be validated. In other words, schemas are used by parsers to verify that an XML document conforms to a specific markup language. To validate an XML document against one or more schemas, you need to specify which schemas to use. You do so by identifying the schemas' locations, using the schemaLocation attribute, which is an XML schema-instance attribute. The XML document uses this attribute to declare the location of the one schema it's based on.

**Using schemaLocation with XML documents Code View:**

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

*<purchaseOrder orderDate="2003-09-22" xmlns="http://www.Durga-jobs.com/jwsbook" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.Durga-jobs.com/jwsbook*

*http://www.Durga-jobs.com/jwsbook/po.xsd"> <accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<shipAddress>*

*<name>AMAZON.COM</name>*

*<street>1850 Mercer Drive</street>*

*<city>Lexington</city>*

*<state>KY</state>*

*<zip>40511</zip>*

*</shipAddress>*

*<billAddress>*

*<name>Amazon.com</name>*

*<street>1516 2nd Ave</street>*

*<city>Seattle</city>*

*<state>WA</state>*

*<zip>90952</zip>*

*</billAddress>*

*<book>*

*<title>J2EE Web Services</title>*

*<quantity>300</quantity>*

*<wholesale-price>24.99</wholesale-price>*

*</book>*

*<total>8997.00</total>*

*</purchaseOrder>*

The second namespace declared in, xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" is the XML schema-instance namespace, which is defined by the XML schema specification.

The XML schema specification explicitly defines a few attributes belonging to this namespace, which can be used in XML documents, including the xsi:schemaLocation attribute.

Another important attribute from the XML schema-instance namespace is xsi:type. The xsi:schemaLocation attribute helps an XML processor locate the actual physical schema document used by the XML instance. Each schema is listed in an xsi:schemaLocation attribute as a namespace-location pair, which associates a namespace with a physical URL.

In the above, the Durga-jobs namespace, "http://www.Durga-jobs.com/jwsbook", is associated with a schema file located at Durga-jobs Books' Web site. You can use xsi:schemaLocation to point at several schemas if you need to.

**For example, we can add the schema location for the XML schema-instance, as below.**

**Declaring Multiple Schema Locations**

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

*<purchaseOrder orderDate="2003-09-22" xmlns="http://www.Durga-jobs.com/jwsbook" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.Durga-jobs.com/jwsbook http://www.Durga-jobs.com/jwsbook/po.xsd http://www.w3.org/2001/XMLSchema-instance http://www.w3.org/2001/XMLSchema.xsd">*

You use white space to separate the namespace and the location URL in each namespace-location set—and to separate namespace-location pairs from each other. For readability, it's a good idea to use more white space to separate sets than to separate each namespace from its location.

You don't actually need to specify the XML schema-instance schema location,[3] because it must be supported natively by any XML schema validating parser, but you should list any other schemas used in an XML document. [3] Whether you should is open to interpretation.

For example, declaring the location of the XML Schema-Instance works with the Apache Xerces-J's SAX parser but not with Altova's XMLSpy (version 5, release 3). For the schemas identified by xsi:schemaLocation to be useful, they must explicitly define themselves as belonging to one of the namespaces identified in the XML instance document.

In this case the schema, belongs to the Durga-jobs Books namespace, "http://www.Durga-jobs.com/jwsbook", the same namespace specified by the instance document. A schema can be located on the Internet, as the Durga-jobs Books schema in is, or on a local hard drive. When using a local schema, specify the location relative to the directory in which the XML document is located.

**For example, shows a schema that's in the same local directory as the XML instance.**

**Pointing to a Schema on a Local File System**

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

*<purchaseOrder orderDate="2003-09-22" xmlns="http://www.Durga-jobs.com/jwsbook" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.Durga-jobs.com/jwsbook po.xsd"> <accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

It's important to note that the xsi:schemaLocation attribute is considered a "hint" by the XML schema specification, which means that XML parsers are not required to use the schema identified by xsi:schemaLocation, but a good parser will, and some, like Xerces-J, allow you to override the location identified by the xsi:schemaLocation attribute programmatically—useful if you want to avoid downloading the schema every time an XML document based on it is parsed; you can use a cached copy instead of the original. The xsi:schemaLocation attribute is usually declared in the root element of an XML document, but it doesn't have to be. You can declare it later in the document, as long as it's in the scope of the elements it applies to.

**Advanced XML Schema**

The key goal of Web services is interoperability, so choosing technologies and standards like XML, SOAP, and WSDL, which are supported by the majority of platforms, is critical. XML is the foundation of Web service interoperability, but even XML can trip you up if you're not careful, particularly the more advanced XML schema types. The painful truth is that XML schema is still new, and some Web service platforms do not support all of its features. That said, according to the WS-I Basic Profile 1.0, Web services must support all of the XML schema features, including those covered in this "Advanced" section.

**Inheritance of Complex Types**

XML schema supports type inheritance much as object-oriented programming languages do, but XML schema inheritance is actually more comprehensive than in most object-oriented languages. Unfortunately, the richness of XML schema inheritance can cause interoperability headaches.

Many Web service platforms map XML schema types to native primitive types, structures, and objects so that developers can manipulate XML data using constructs native to their programming environment. For example, JAX-RPC maps some of the XML schema built-in types to Java primitives, and basic complex types to Java beans.

JAX-RPC can map most derived complex types to Java beans, but not all. Similar limitations are found in other platforms like .NET and SOAP::Lite for Perl.

Most object-oriented languages do not support the full scope of inheritance defined by the XML schema specification. For this reason, you should use type inheritance in schemas with care.

**Complex types can use two types of inheritance:**

* extension and
* restriction.

Both allow you to derive new complex types from existing complex types.

Extension broadens a derived type by adding elements or attributes not present in the base type, while restriction narrows a derived type by omitting or constraining elements and attributes defined by the base type.

**Extension**

An extension type inherits the elements and attributes of its base type, and adds new ones.

**For example, we could redefine the USAddress type to be an extension of a base type called Address as shown below.**

**Using XML Schema Inheritance**

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

*<schema targetNamespace="http://www.Durga-jobs.com/jwsbook" xmlns:mh="http://www.Durga-jobs.com/jwsbook" xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">*

*<element name="address" type="mh:Address"/>*

*<complexType name="Address">*

*<sequence>*

*<element name="name" type="string"/>*

*<element name="street" type="string" maxOccurs="unbounded"/>*

*<element name="city" type="string"/>*

*<element name="country" type="string"/>*

*</sequence>*

*<attribute name="category" type="string" default="business"/>*

*</complexType>*

*<complexType name="USAddress">*

*<complexContent>*

*<extension base="mh:Address">*

*<sequence>*

*<element name="state" type="string"/>*

*<element name="zip" type="string"/>*

*</sequence>*

*</extension>*

*</complexContent>*

*</complexType>*

*...*

*</schema>*

The complexType and extension elements tell us that USAddress extends Address. It adds the state and zip elements, so that the USAddress type has a total of six elements (name, street, city, state, zip, and country).

The base type Address defined can be used to create other derived types as well.

For example, we could extend it to define a United Kingdom address type, UKAddress.

**A UK Address Type Extends the Address Type**

*<complexType name="UKAddress" >*

*<complexContent>*

*<extension base="mh:Address">*

*<sequence>*

*<element name="postcode" type="string"/>*

*</sequence>*

*</extension>*

*</complexContent>*

*</complexType>*

We now have two types derived from the Address type, USAddress and UKAddress, which capture the addressing proper to their respective postal systems.

Actually, many UK addresses may not have a city, but we will ignore that detail in this example.

**Restriction**

Restriction is very easy to understand. You simply redefine or omit those elements and attribute that change, and list all the other elements and attributes exactly as they were in the base type. For example, we can create a USAddress type that omits the city and state elements, as shown. (If you have a zip code you don't need a city and state, because any zip code can be cross-referenced to a specific city and state.)

**An Extension of the USAddress Type Defined**

*<complexType name="BriefUSAddress">*

*<complexContent>*

*<restriction base="mh:USAddress">*

*<sequence>*

*<element name="name" type="string"/>*

*<element name="street" type="string"/>*

*<element name="zip" type="string"/>*

*</sequence>*

*<attribute name="category" type="string" default="business"/>*

*</restriction>*

*</complexContent>*

*</complexType>*

In this example, the derived type, BriefUSAddress, contains the name, street, and zip elements, but not the city, state, and country elements, because the schema simply omits them. In addition we have redefined the occurrence constraints on the street element so that it may occur only once (recall that the default values of maxOccurs and minOccurs are both "1"). Compare BriefUSAddress to the Address base type in, which defined the street element with a maxOccurs equal to "unbounded".

While the above paragraph is correct, there are some important limits on what you can do: You cannot omit an element from a restriction unless the parent type declared it to be optional(minOccurs="0"). In addition, the derived type's occurrence constraints cannot be less strict than those of its base type.

For example, you cannot constrain an element to minOccurs="0" and maxOccurs="4" in the child if the parent's element is defined as minOccurs="1" and maxOccurs="2". The restricted occurrence attributes must fall within the boundaries defined by the parent type.

For the BriefUSAddress to work, we will need to redefine the USAddress type to make the city and state elements optional (set minOccurs="0"); if we don't, the parser will report an error. The necessity of repeating all the elements and attributes, even if they don't change, makes restriction a bit cumbersome, but it's the only logical way of indicating which elements and attributes are omitted or constrained. While restriction is useful, it's used less than extension because it doesn't map as well to programming languages.

For this reason, it's risky to use restriction when defining complex types in your XML documents.

**Importing and Including Schemas**

You can combine schemas using two different elements, include and import.

An import allows you to combine schemas from different namespaces, while an include lets you combine schemas from the same namespace.

**Importing**

A schema may import types from other schemas, allowing more modular schema design and type reuse.

For example, we can define a separate schema and namespace for all the types related to mailing addresses: Address, USAddress, UKAddress, BriefUSAddress, USZipCode, and USState. This schema would define the complete Address Markup Language for Durga-jobs Books.

**The Address Markup Schema Code View:**

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

<schema targetNamespace="http://www.Durga-jobs.com/addr" xmlns:addr="http://www.Durga-jobs.com/addr" xmlns="http://www.w3.org/2001/XMLSchema">

<element name="address" type="addr:Address"/>

<simpleType name="USZipCode">

<restriction base="string">

<pattern value="[0-9]{5}(-[0-9]{4})?"/>

</restriction>

</simpleType>

<simpleType name="USState">

<restriction base="string">

<enumeration value="AK"/>

<!-- Alaska -->

<enumeration value="AL"/>

<!-- Alabama -->

<enumeration value="AR"/>

<!-- Arkansas -->

<!-- and so on -->

</restriction>

</simpleType>

<complexType name="Address" abstract="true">

<sequence>

<element name="name" type="string"/>

<element name="street" type="string" maxOccurs="unbounded"/>

<element name="city" type="string"/>

<element name="country" type="string"/>

</sequence>

<attribute name="category" type="string" default="business"/>

</complexType>

<complexType name="USAddress" final="extension">

<complexContent>

<extension base="addr:Address">

<sequence>

<element name="state" type="addr:USState"/>

<element name="zip" type="addr:USZipCode"/>

</sequence>

</extension>

</complexContent>

</complexType>

<complexType name="UKAddress">

<complexContent>

<extension base="addr:Address">

<sequence>

<element name="postcode" type="string"/>

</sequence>

</extension>

</complexContent>

</complexType>

<complexType name="BriefUSAddress">

<complexContent>

<restriction base="addr:USAddress">

<sequence>

<element name="name" type="string"/>

<element name="street" type="string"/>

<element name="zip" type="addr:USZipCode"/>

</sequence>

<attribute name="category" type="string" default="business"/>

</restriction>

</complexContent>

</complexType>

</schema>

The targetNamespace of the Address Markup schema is "http://www.Durga-jobs.com/jwsbook/ADDR", which is a separate namespace from that of the purchase-order elements. Because the PurchaseOrdertype depends on the Address type, we'll need to import the Address Markup schema into the Purchase Order schema as shown.

**Importing a Schema**

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

<schema targetNamespace="http://www.Durga-jobs.com/jwsbook/PO" xmlns:po="http://www.Durga-jobs.com/jwsbook/PO" xmlns:addr="http://www.Durga-jobs.com/jwsbook/ADDR" xmlns="http://www.w3.org/2001/XMLSchema">

<import namespace="http://www.Durga-jobs.com/jwsbook/ADDR" schemaLocation="http://www.Durga-jobs.com/jwsbook/addr.xsd"/>

<element name="purchaseOrder" type="po:PurchaseOrder"/>

<simpleType name="Total">

<restriction base="float">

<minInclusive value="0.00"/>

<maxExclusive value="100000.00"/>

<pattern value="[0-9]+\.[0-9]{2}"/>

</restriction>

</simpleType>

<complexType name="PurchaseOrder">

<sequence>

<element name="accountName" type="string"/>

<element name="accountNumber" type="unsignedShort"/>

<element name="shipAddress" type="addr:Address"/>

<element name="billAddress" type="addr:Address"/>

<element name="book" type="po:Book"/>

<element name="total" type="po:Total"/>

</sequence>

<attribute name="orderDate" type="date"/>

</complexType>

<complexType name="Book">

<sequence>

<element name="title" type="string"/>

<element name="quantity" type="unsignedShort"/>

<element name="wholesale-price" type="float"/>

</sequence>

</complexType>

</schema>

The import mechanism enables you to combine schemas to create larger, more complex schemas. It's very useful when you see that some aspects of a schema, such as the address types, are reusable and need their own namespace and schema. The imported namespace needs to be assigned a prefix before we can use it. In this case, it's assigned the prefix addr in the root schema element.

**Including**

In addition to the import element, there is another way of combining schemas called include, which can be used only to combine schemas with exactly the same targetNamespace. Including is useful when a schema becomes large and difficult to maintain. The Purchase Order schema has not become that unwieldy, but just as an example, we could place the definitions of the Total and Book types into a separate schema, then use an include element to combine them with the Purchase Order schema. shows a schema document for the Total and Book elements, which we'll soon include in the Purchase Order schema.

The Book and Total Schema

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

<schema targetNamespace="http://www.Durga-jobs.com/jwsbook/PO" xmlns:po="http://www.Durga-jobs.com/jwsbook/PO" xmlns="http://www.w3.org/2001/XMLSchema">

<simpleType name="Total">

<restriction base="float">

<minInclusive value="0.00"/>

<maxExclusive value="100000.00"/>

<pattern value="[0-9]+\.[0-9]{2}"/>

</restriction>

</simpleType>

<complexType name="Book">

<sequence>

<element name="title" type="string"/>

<element name="quantity" type="unsignedShort"/>

<element name="wholesale-price" type="float"/>

</sequence>

</complexType>

</schema>

Here the Book and Total types have been placed in their own schema document—but notice that the targetNamespace is the same as in the Purchase Order schema. We can combine these two schemas using an include statement. shows the use of both import and include.

**Using Import and Include Together Code View: Scroll / Show All**

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

<schema targetNamespace="http://www.Durga-jobs.com/jwsbook/PO" xmlns:po="http://www.Durga-jobs.com/jwsbook/PO" xmlns:addr="http://www.Durga-jobs.com/jwsbook/ADDR" xmlns="http://www.w3.org/2001/XMLSchema">

<include schemaLocation="http://www.Durga-jobs.com/jwsbook/po.xsd"/>

<import namespace="http://www.Durga-jobs.com/jwsbook/ADDR" schemaLocation="http://www.Durga-jobs.com/jwsbook/addr.xsd"/>

<element name="purchaseOrder" type="po:PurchaseOrder"/>

<complexType name="PurchaseOrder">

<sequence>

<element name="accountName" type="string"/>

<element name="accountNumber" type="unsignedShort"/>

<element name="shipAddress" type="addr:Address"/>

<element name="billAddress" type="addr:Address"/>

<element name="book" type="po:Book"/>

<element name="total" type="po:Total"/>

</sequence>

<attribute name="orderDate" type="date"/>

</complexType>

</schema>

Notice that we don't specify the namespace of the included schema, because it's expected to match the targetNamespace of the schema, doing the including.