This article gives a basic overview of the building blocks underlying XML Schemas and how to use them. It covers:

* Schema Overview
* Elements
* Cardinality
* Simple Types
* Complex Types
* Compositors
* Reuse
* Attributes
* Mixed Element Content

**Overview**

First, look at what an XML schema is. 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 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 into any element or attribute.

**A Note About Standards**http://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=yhttp://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=yhttp://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=yhttp://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=yhttp://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=yhttp://www.codeguru.com/hqxapi/it?ids=2014112201123740829165001&ca=y&coo=y

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. (*You have to dig quite deep before you notice the difference*.) An XSD schema is itself an XML document; there is even an a XSD schema to describe the XSD standard.

There are also a number of other standards, but their take up has been patchy at best.

The XSD standard has evolved over a number of years, and is controlled by the W3C. It is extremely comprehensive, and as a result has become rather complex. For this reason, it is a good idea to make use of design tools when working with XSDs ([See XML Studio, a FREE XSD development too](http://www.liquid-technologies.com/LiquidXMLStudio.aspxv)l), also when working with XML documents programmatically [XML Data Binding](http://www.liquid-technologies.com/Products_XMLDataBinding_ReasonsToUse.aspx) is a much easier way to manipulate your documents (a object-oriented approach; see Liquid XML Data Binding).

The remainder of this tutorial guides you through the basics of the XSD standard, things you should really know even if you're using a design tool like [Liquid XML Studio.](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)

**Elements**

Elements are the main building block of any XML document; they contain the data and determine the structure of the document. An element can be defined within an XML Schema (XSD) as follows:

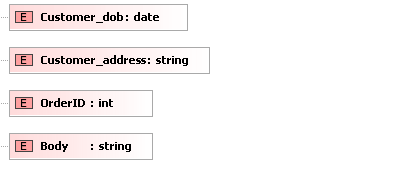
1. <xs:element name="x" *type="y"*/>

An element definition within the XSD must have a name property; this is the name that will appear in the XML document. 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:date (see [the XSD standard for a complete list](http://www.w3.org/TR/xmlschema-2/#built-in-datatypes)). You also can create a user-defined type by using the <xs:simple type> and <xs:complexType> tags, but more on these later.

If you have set the type property for an element in the XSD, the corresponding value in the XML document must be in the correct format for its given type. (Failure to do this will cause a validation error.) Examples of simple elements and their XML are below:

|  |  |
| --- | --- |
| **Sample XSD** | **Sample XML** |
| 1. <xs:element name="Customer\_dob" 2. type="xs:date"/> | 1. <Customer\_dob> 2. 2000-01-12T12:13:14Z 3. </Customer\_dob> |
| 1. <xs:element name="Customer\_address" 2. type="xs:string"/> | 1. <Customer\_address> 2. 99 London Road 3. </Customer\_address> |
| 1. <xs:element name="OrderID" 2. type="xs:int"/> | 1. <OrderID> 2. 5756 3. </OrderID> |
| 1. <xs:element name="Body" 2. type="xs:string"/> | 1. <Body> 2. (a type can be defined as 3. a string but not have any 4. content; this is not true 5. of all data types, however). 6. </Body> |

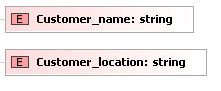
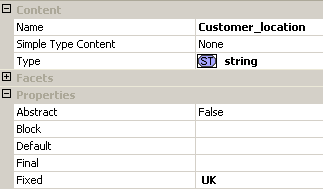
The previous XSD definitions are shown graphically in [Liquid XML Studio](http://www.liquid-technologies.com/LiquidXMLStudio.aspx) as follows:

[[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)The value the element takes in the XML document can further be affected by using the *fixed* and *default* properties.

***Default*** means that, if no value is specified in the XML document, 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. (In fact, it's illegal to do so.)

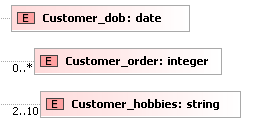
1. <xs:element name="Customer\_name" type="xs:string" default="unknown"/>
2. <xs:element name="Customer\_location" type="xs:string" fixed=" UK"/>

[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)      [[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)**Cardinality**

Specifying how many times an element can appear is referred to as *cardinality*, and is specified by using the minOccurs and maxOccurs attributes. In this way, an element can be mandatory, optional, or appear many times. minOccurs can be assigned any non-negative integer value (for example: 0, 1, 2, 3... and so forth), and maxOccurs can be assigned any non-negative integer value or the string constant *"unbounded"*, meaning no maximum.

The default values for minOccurs and maxOccurs is 1. So, if both the minOccurs and maxOccurs attributes are absent, as in all the previous examples, the element must appear once and once only.

|  |  |
| --- | --- |
| **Sample XSD** | **Description** |
| 1. <xs:element name="Customer\_dob" 2. type="xs:date"/> | If you don't specify minOccurs or maxOccurs, the default values of 1 are used, so in this case there has to be one and only one occurrence of Customer\_dob |
| 1. <xs:element name="Customer\_order" 2. type="xs:integer" 3. minOccurs ="0" 4. maxOccurs="unbounded"/> | Here, a customer can have any number of Customer\_orders (even 0) |
| 1. <xs:element name="Customer\_hobbies" 2. type="xs:string" 3. minOccurs="2" 4. maxOccurs="10"/> | In this example, the element Customer\_hobbies must appear at least twice, but no more than 10 times |

[[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)**Simple Types**

So far, you have touched on a few of the built-in data types xs:string, xs:integer, and xs:date. But, you also can define your own types by modifying existing ones. Examples of this would be:

* Defining an ID; this may be an integer with a max limit.
* A PostCode or Zip code could be restricted to ensure it is the correct length and complies with a regular expression.
* A field may have a maximum length.

Creating your own types is coved more thoroughly in the next section.

**Complex Types**

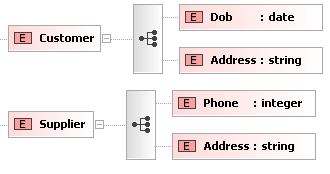
A complex type is a container for other element definitions; this allows you to specify which child elements an element can contain. This allows you to provide some structure within your XML documents.

Have a look at these simple elements:

1. <xs:element name="Customer" type="xs:string"/>
2. <xs:element name="Customer\_dob" type="xs:date"/>
3. <xs:element name="Customer\_address" type="xs:string"/>
5. <xs:element name="Supplier" type="xs:string"/>
6. <xs:element name="Supplier\_phone" type="xs:integer"/>
7. <xs:element name="Supplier\_address" type="xs:string"/>

You 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". By the same token, "Supplier\_phone" and "Supplier\_address" belong to a parent element "Supplier". You can therefore re-write this in a more structured way:

1. <xs:element name="Customer">
2. <xs:complexType>
3. <xs:sequence>
4. <xs:element name="Dob" type="xs:date" />
5. <xs:element name="Address" type="xs:string" />
6. </xs:sequence>
7. </xs:complexType>
8. </xs:element>
10. <xs:element name="Supplier">
11. <xs:complexType>
12. <xs:sequence>
13. <xs:element name="Phone" type="xs:integer"/>
14. <xs:element name="Address" type="xs:string"/>
15. </xs:sequence>
16. </xs:complexType>
17. </xs:element>

[[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)  
The previous XSD shown graphically using Liquid XML Studio](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)

**Example XML**

1. <Customer>
2. <Dob> 2000-01-12T12:13:14Z </Dob>
3. <Address>
4. 34 thingy street, someplace, sometown, w1w8uu
5. </Address>
6. </Customer>
8. <Supplier>
9. <Phone>0123987654</Phone>
10. <Address>
11. 22 whatever place, someplace, sometown, ss1 6gy
12. </Address>
13. </Supplier>

**What's Changed?**

Look at this in detail.

* You created a definition for an element called "Customer".
* Inside the <xs:element> definition, you added a <xs:complexType>. This is a container for other <xs:element> definitions, allowing you to build a simple hierarchy of elements in the resulting XML document.
* Note that the contained elements for "Customer" and "Supplier" do not have a type specified because 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 because it is implied from its position within the parent element "Customer" or "Supplier".

So, in English, this is saying you can have an XML document that contains a <Customer> element that must have teo child elements. <Dob> and <Address>.

**Compositors**

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

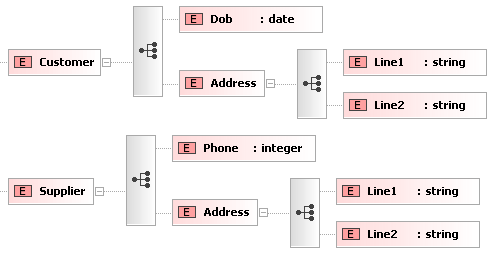
|  |  |
| --- | --- |
| **Compositor** | **Description** |
| Sequence | The child elements in the XML document MUST appear in the order they are declared in the XSD schema. |
| Choice | Only one of the child elements described in the XSD schema can appear in the XML document. |
| All | The child elements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

**Notes**

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

One step further: 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:

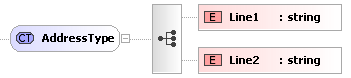
1. <xs:element name="Customer">
2. <xs:complexType>
3. <xs:sequence>
4. <xs:element name="Dob" type="xs:date" />
5. <xs:element name="Address">
6. <xs:complexType>
7. <xs:sequence>
8. <xs:element name="Line1" type="xs:string" />
9. <xs:element name="Line2" type="xs:string" />
10. </xs:sequence>
11. </xs:complexType>
12. </xs:element>
13. </xs:sequence>
14. </xs:complexType>
15. </xs:element>
17. <xs:element name="Supplier">
18. <xs:complexType>
19. <xs:sequence>
20. <xs:element name="Phone" type="xs:integer" />
21. <xs:element name="Address">
22. <xs:complexType>
23. <xs:sequence>
24. <xs:element name="Line1" type="xs:string" />
25. <xs:element name="Line2" type="xs:string" />
26. </xs:sequence>
27. </xs:complexType>
28. </xs:element>
29. </xs:sequence>
30. </xs:complexType>
31. </xs:element>

[[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)This is much better, but you now have two definitions for address, which are the same.

**Re-Use**

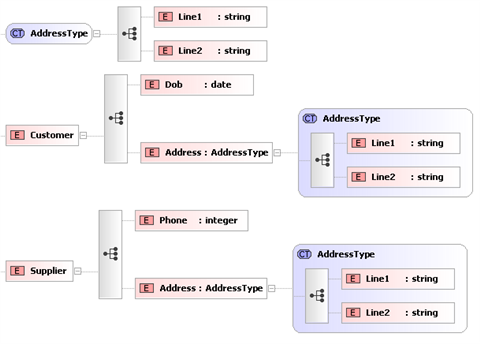
It would make much more sense to have one definition of "Address" that could be used by both customer and supplier. You can do this by defining a complexType independently of an element, and giving it a unique name:

1. <xs:complexType name="AddressType">
2. <xs:sequence>
3. <xs:element name="Line1" type="xs:string"/>
4. <xs:element name="Line2" type="xs:string"/>
5. </xs:sequence>
6. </xs:complexType>

[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)

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.

1. <xs:element name="Customer">
2. <xs:complexType>
3. <xs:sequence>
4. <xs:element name="Dob" type="xs:date"/>
5. <xs:element name="Address" type="AddressType"/>
6. </xs:sequence>
7. </xs:complexType>
8. </xs:element>
10. <xs:element name="supplier">
11. <xs:complexType>
12. <xs:sequence>
13. <xs:element name="address" type="AddressType"/>
14. <xs:element name="phone" type="xs:integer"/>
15. </xs:sequence>
16. </xs:complexType>
17. </xs:element>

[](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)

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 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**

1. <Customer>
2. <Dob> 2000-01-12T12:13:14Z </Dob>
3. <Address>
4. <Line1>34 thingy street, someplace</Line1>
5. <Line2>sometown, w1w8uu </Line2>
6. </Address>
7. </Customer>
9. <Supplier>
10. <Phone>0123987654</Phone>
11. <Address>
12. <Line1>22 whatever place, someplace</Line1>
13. <Line2>sometown, ss1 6gy </Line2>
14. </Address>
15. </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.

**Attributes**

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

1. <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 specifies whether the attribute is optional or mandatory.

So, the following are equivalent:

1. <xs:attribute name="ID" type="xs:string"/>
2. <xs:attribute name="ID" type="xs:string" use="optional"/>

The previous XSD definitions are shown graphically in [Liquid XML Studio](http://www.liquid-technologies.com/LiquidXMLStudio.aspx) as follows:

[[http://www.codeguru.com/images/article/13529/img4.png](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)](http://www.liquid-technologies.com/LiquidXMLStudio.aspx)To specify that an attribute must be present, use = "required" *(note that use may also be set to "prohibited", but you'll come to that later)*.

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

|  |  |
| --- | --- |
| **Sample XSD** | **Sample XML** |
| 1. <xs:element name="Order"> 2. <xs:complexType> 3. <xs:attribute name="OrderID" 4. type="xs:int"/> 5. </xs:complexType> 6. </xs:element> | 1. <Order OrderID="6"/>   *or*   1. <Order/> |
| 1. <xs:element name="Order"> 2. xs:complexType> 3. <xs:attribute name="OrderID" 4. type="xs:int" 5. use="optional"/> 6. </xs:complexType> 7. </xs:element> | 1. <Order OrderID="6"/>   *or*   1. <Order/> |
| 1. <xs:element name="Order"> 2. <xs:complexType> 3. <xs:attribute name="OrderID" 4. type="xs:int" 5. use="required"/> 6. </xs:complexType> 7. </xs:element> | 1. <Order OrderID="6"/> |

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

**Mixed Element Content**

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

1. <xs:element name="MarkedUpDesc">
2. <xs:complexType mixed="true">
3. <xs:sequence>
4. <xs:element name="Bold" type="xs:string" />
5. <xs:element name="Italic" type="xs:string" />
6. </xs:sequence>
7. </xs:complexType>
8. </xs:element>

A sample XML document could look like this.

1. <MarkedUpDesc>
2. This is an <Bold>Example</Bold> or
3. <Italic>Mixed</Italic> Content,
4. Note there are elements mixed in with the elements data.
5. </MarkedUpDesc>

**History**

Keep a running update of any changes or improvements you've made here.

This section covers conventions and recommendations when designing your schemas.

* When to use Elements or Attributes
* Mixed Element Content
* Conventions

**When to Use Elements or Attributes**

There is often some confusion over when to use an element or an attribute. Some people say that elements describe data and attributes describe the metadata; another way to look at it is that attributes are used for small pieces of data such as order IDs, but really it is personal taste that dictates when to use an attribute. Generally, it is best to use a child element if the information feels like data. Some of the problems with using attributes are:

* Attributes cannot contain multiple values (child elements can).
* Attributes are not easily expandable (to incorporate future changes to the schema).
* Attributes cannot describe structures (child elements can).

lf you use attributes as containers for data, you end up with documents that are difficult to read and maintain. Try to use elements to describe data. What I am trying to say here is that metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

**Mixed Element Content**

Mixed content is something you should try to avoid as much as possible. It is used heavily on the web in the form of xHtml, but it has many limitations. It is difficult to parse and it can lead to unforeseen complexity in the resulting data. XML Data Binding has limitations associated with it making it difficult to manipulate such documents.

**Conventions**

* All Element and attributes should use UCC camel case, for example PostalAddress, avoid hyphens, spaces, or other syntax.
* Readability is more important than tag length. There is always a line to draw between document size and readability; wherever possible, favor readability.
* Try to avoid abbreviations and acronyms for element, attribute, and type names. Exceptions should be well known within your business area, for example ID (Identifier), and POS (Point of Sale).
* Postfix new types with the name 'Type'. eg AddressType, USAddressType.
* Enumerations should use names, not numbers, and the values should be UCC camel case.
* Names should not include the name of the containing structure; for example, CustomerName should be Name within the sub element Customer.
* Only produce complexTypes or simpleTypes for types that are likely to be re-used. If the structure exists only in one place, define it inline with an anonymous complexType.
* Avoid the use of mixed content.
* Only define root level elements if the element is capable of being the root element in an XML document.
* Use consistent name space aliases:
  + **xml**: Defined in the XML standard
  + **xmlns**: Defined in Name spaces in the XML standard
  + **xs**: http://www.w3.org/2001/XMLSchema
  + **xsi**: http://www.w3.org/2001/XMLSchema-instance
* Try to think about versioning early in your schema design. If it's important for a new version of a schema to be backwardly compatible, 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, consider adding any and all anyAttribute entries to the end of your definitions. See [Versioning recommendations.](http://www.w3.org/TR/xmlschema-guide2versioning/)
* 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 (if not more verbose).

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 development languages, you would call this inheritance or sub classing. The same concepts also exist in the XSD standard. This allows you to take an existing type definition and extend it. You also can restrict an existing type (although this behavior has no real parallel in most development languages).

* Extending a ComplexType
* Restricting an Existing ComplexType
* Use of Extended/Restricted Types
* Extending Simple Types (Union, List, Restriction)

**Extending an Existing ComplexType**

It is possible to take an existing <xs:complexType> and extend it. See how this may be useful by looking at an example.

Looking at the AddressType that you defined earlier in Part 1), assume your company has now gone international and you need to capture country-specific addresses. In this case, you need specific information for UK addresses (County and Postcode), and for US addresses (State and ZipCode).

So, you can take your existing definition of address and extend it as follows:

1. <xs:complexType name="UKAddressType">
2. <xs:complexContent>
3. **<xs:extension base="AddressType">**
4. <xs:sequence>
5. <xs:element name="County" type="xs:string"/>
6. <xs:element name="Postcode" type="xs:string"/>
7. </xs:sequence>
8. </xs:extension>
9. </xs:complexContent>
10. </xs:complexType>
12. <xs:complexType name="USAddressType">
13. <xs:complexContent>
14. **<xs:extension base="AddressType">**
15. <xs:sequence>
16. <xs:element name="State" type="xs:string"/>
17. <xs:element name="Zipcode" type="xs:string"/>
18. </xs:sequence>
19. </xs:extension>
20. </xs:complexContent>
21. </xs:complexType>

This is clearer when viewed graphically. But basically, it is saying that you are defining a new <xs:complexType> called "USAddressType". This extendeds the existing type "AddressType" and adds to it a sequence containing the elements "State" and "Zipcode".

There are two new things here the <xs:extension> element and the <xs:complexContent> element. I'll get to these shortly.

You now can use these new types as follows:

1. <xs:element name="UKAddress" type="UKAddressType"/>
2. <xs:element name="USAddress" type="USAddressType"/>

Some sample XML for these elements may look like this.

1. <UKAddress>
2. <Line1>34 thingy street</Line1>
3. <Line2>someplace</Line2>
4. **<County>somerset/County>**
5. **<Postcode>w1w8uu</Postcode>**
6. </UKAddress>

or

1. <USAddress>
2. <Line1>234 Lancaster Av</Line1>
3. <Line2>Smallville</Line2>
4. **<State>Florida</State>**
5. **<Zipcode>34543</Zipcode>**
6. </USAddress>

The last example showed how to take an existing <xs:complexType> definition and extend it to create new types. The new construct <xs:extension> indicates that you are extending an existing type, and specifies the type itself. But, there is another option here; instead of adding to the type, you could restrict it.

**Restricting an Existing ComplexType**

Taking the same AddressType example, you can create a new type called "InternalAddressType". Assume that "InternalAddressType" only needs Address->Line1.

1. <xs:complexType name="InternalAddressType">
2. <xs:complexContent>
3. **<xs:restriction base="AddressType">**
4. <xs:sequence>
5. <xs:element name="Line1" type="xs:string" />
6. </xs:sequence>
7. </xs:restriction>
8. </xs:complexContent>
9. </xs:complexType>

You are defining a new type, "InternalAddressType". The <xs:restriction> element says you are restricting the existing type "AddressType", and you are only allowing the existing child element "Line1" to be used in this new definition.

**Note:** Because you 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 also must be enclosed in the same compositor (in this case a sequence) and appear in the same order.

You can now use this new type as follows:

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

Some sample XML for this element may look like this.

1. <InternalAddressType>
2. <Line1>Desk 4, Second Floor/<Line1>
3. </InternalAddressType>

**Note:** The <xs:complexContent> element is just a container for the extension or restriction. You largely can ignore it for now.

**Use of Extended/Restricted Types**

You have just seen how you 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 examples above, you created three new types (UKAddressType, USAddressType, and InternalAddressType), all based on AddressType.

So, if you have an element that specifies it's of type UKAddressType, that is what must appear in the XML document. But, if an element specifies it's of type "AddressType", any of the four 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 cannot 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), you have to let the parser know which type you're using. This is done in the XML document using the xsi:type attribute. Look at an example.

1. <xs:element name="Person">
2. <xs:complexType>
3. <xs:sequence>
4. <xs:element name="Name" type="xs:string" />
5. **<xs:element name="HomeAddress" type="AddressType" />**
6. </xs:sequence>
7. </xs:complexType>
8. </xs:element>

This sample XML is the kind of thing you would expect to see.

1. <?xml version="1.0"?>
2. <Person>
3. <Name>Fred</Name>
4. **<HomeAddress>**
5. **<Line1>22 whatever place, someplace</Line1>**
6. **<Line2>sometown, ss1 6gy </Line2>**
7. **</HomeAddress>**
8. </Person>

But, the following is also valid.

1. <?xml version="1.0"?>
2. <Person **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**>
3. <Name>Fred</Name>
4. <HomeAddress **xsi:type="USAddressType"**>
5. <Line1>234 Lancaseter Av</Line1>
6. <Line2>SmallsVille</Line2>
7. **<State>Florida</State>**
8. **<Zipcode>34543</Zipcode>**
9. </HomeAddress>
10. </Person>

Look at that in more detail.

* You 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 aliased by "xsi" that 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.

But, more about namespaces in the next section.

**Extending Simple Types**

There are three ways in which a simpleType can be extended: Restriction, List, or Union. The most common is Restriction, but I will cover the other two as well.

**Restriction**

Restriction is a way to constrain an existing type definition. You can apply a restriction to the built-in data types xs:string, xs:integer, xs:date, and so forth or ones you create yourself.

Here, you define a restriction the existing type "string". You apply a regular expression to it, to limit the values it can take.

1. <xs:simpleType name="LetterType">
2. <xs:restriction base="xs:string">
3. <xs:pattern value="[a-zA-Z]"/>
4. </xs:restriction>
5. </xs:simpleType>

Go through this line by line.

1. A <simpleType> tag is used to define your new type. You must give the type a unique name, in this case "LetterType".
2. You are restricting an existing type, so the tag is <restriction>. (You also can extend an existing type, but more about this later.) You are basing your new type on a string, so type="xs:string"
3. You are applying a restriction in the form of a Regular expression; this is specified by 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**. For a complete list, see the[XSD Standard](http://www.w3.org/TR/xmlschema-2/#rf-facets), but to give you an idea, here are a few to get you started.

|  |  |  |
| --- | --- | --- |
| **Overview** | **Syntax** | **Syntax Explained** |
| This specifies the minimum and maximum length allowed  Must be 0 or greater | <xs:minLength value="3"> <xs:maxLength value="8"> | In this example, the length must be between 3 and 8 |
| The lower and upper range for numerical values  The value must be less than or equal to, greater than or equal to | <xs:minInclusive value="0"> <xs:maxInclusive value="10"> | The value must be between 0 and 10 |
| The lower and upper range for numerical values  The value must be less than or greater than | <xs:minExclusive value="0"> <xs:maxExclusive value="10"> | The value must be between 1 and 9 |
| The exact number of characters allowed | *<xs:length value="30">* | The length must not be more than 30 |
| Exact number of digits allowed | *<xs:totalDigits value="9">* | Cannot have more than 9 digits |
| A list of values allowed | <xs:enumeration value="Hippo"/> <xs:enumeration value="Zebra"/> <xs:enumeration value="Lion"/> | The only permitted values are Hippo, Zebra, or Lion |
| The number of decimal places allowed (must be >= 0) | <xs:fractionDigits value="2"/> | The value has to be to 2 d.p. |
| This defines how whitespace will be handled. Whitespace is line feeds, carriage returns, tabs, spaces, and so on. | <xs:whitespace value=   "preserve"/> <xs:whitespace value=   "replace"/> <xs:whitespace value=   "collapse"/> | *Preserve*: Keeps whitespaces *Replace*: Replaces all whitespace with a space *Collape*: Replaces whitespace characters with a space. If there are multiple spaces together, they will be reduced to one space. |
| Pattern determines what characters are allowed and in what order. These are regular expressions and there is a complete list at: http://www.w3.org/TR/xmlschema-2/#regexs | <xs:pattern value="[0-999]"/> | [0-999]: One digit only between 0 and 999  [0-99][0-99][0-99]: Three digits, all have to be between 0 and 99  [a-z][0-10][A-Z]: The first digit has to be between a and z, the second digit has to be between 0 and 10, the third digit is between A and Z. These are case sensitive.  [a-zA-Z]: One digit that can be either lower or uppercase A–Z  [123]: One digit that has to be 1, 2, or 3  ([a-z])\*: Zero or more occurrences of a to z  ([q][u])+: Looking for a pair of letters that satisfy the criteria; in this case, a q followed by a u  ([a-z][0-999])+: As above, looking for a pair where the first digit is lowercase and between a and z, and the second digit is between 0 and 999; for example a1, c99, z999, f45  [a-z0-9]{8}: Must be exactly 8 characters in a row and they must be lowercase a to z or number 0 to 9. |

*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 XSD standard.*](http://www.w3.org/TR/xmlschema-0/#SimpleTypeFacets)

**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 (for example, 10, 12, 14), and "SizeByStringNameType" the values *small*, *medium*, and *large*.

1. <xs:simpleType name="SizeByNumberType">
2. <xs:restriction base="xs:positiveInteger">
3. <xs:maxInclusive value="21"/>
4. </xs:restriction>
5. </xs:simpleType>
7. <xs:simpleType name="SizeByStringNameType">
8. xs:restriction base="xs:string">
9. <xs:enumeration value="small"/>
10. <xs:enumeration value="medium"/>
11. <xs:enumeration value="large"/>
12. </xs:restriction>
13. </xs:simpleType>

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

1. <xs:simpleType name="USClothingSizeType">
2. <xs:union memberTypes="SizeByNumberType SizeByStringNameType" />
3. </xs:simpleType>

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

This new type then can 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 is that you 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".

1. <xs:simpleType name="SizesinStockType">
2. <xs:list itemType="SizeByNumberType" />
3. </xs:simpleType>

Namespaces are a mechanism for breaking up your schemas. Until now, you 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 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.

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

* **CommonTypes**: This could contain all your basic types: AddressType, PriceType, PaymentMethodType, and so forth.
* **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 element/s.

This all works fine without namespaces, but if different teams start working on different files, 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.

You can do this by adding the attribute targetNamespace into the schema element in the XSD file; in other words:

1. <?xml version="1.0"?>
2. <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3. **targetNamespace="myNamespace"**>
4. ...
5. </xs:schema>

The value of targetNamespace is just a unique identifier; typically, companies use there URL followed by something to qualify it. In principle, the namespace has no meaning, but some companies have used the URL where the schema is stored because the targetNamespace and some XML parsers will use this as a hint path for the schema targetNamespace="http://www.microsoft.com/CommonTypes.xsd", but the following would be just as 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 the example above, each of the four schema files could have a distinct targetNamespace value.

Look at them in detail.

**CommonTypes.xsd**

1. <?xml version="1.0" encoding="utf-16"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <xs:schema **targetNamespace="http://NamespaceTest.com/CommonTypes"**
5. xmlns:xs="http://www.w3.org/2001/XMLSchema"
6. elementFormDefault="qualified">
8. <xs:complexType name="AddressType">
9. <xs:sequence>
10. <xs:element name="Line1" type="xs:string" />
11. <xs:element name="Line2" type="xs:string" />
12. </xs:sequence>
13. </xs:complexType>
14. <xs:simpleType name="PriceType">
15. <xs:restriction base="xs:decimal">
16. <xs:fractionDigits value="2" />
17. </xs:restriction>
18. </xs:simpleType>
20. <xs:simpleType name="PaymentMethodType">
21. <xs:restriction base="xs:string">
22. <xs:enumeration value="VISA" />
23. <xs:enumeration value="MasterCard" />
24. <xs:enumeration value="Cash" />
25. <xs:enumeration value="Amex" />
26. </xs:restriction>
27. </xs:simpleType>
28. </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 "http://NamespaceTest.com/CommonTypes".

**CustomerTypes.xsd**

1. <?xml version="1.0" encoding="utf-16"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <xs:schema  **xmlns:cmn="http://NamespaceTest.com/CommonTypes"**
5. **targetNamespace="http://NamespaceTest.com/CustomerTypes"**
6. xmlns:xs="http://www.w3.org/2001/XMLSchema"
7. elementFormDefault="qualified">
8. **<xs:import schemaLocation="CommonTypes.xsd"**
9. **namespace="http://NamespaceTest.com/CommonTypes"/>**
10. <xs:complexType name="CustomerType">
11. <xs:sequence>
12. <xs:element name="Name" type="xs:string" />
13. <xs:element name="DeliveryAddress" type="cmn:AddressType" />
14. <xs:element name="BillingAddress" type="cmn:AddressType" />
15. </xs:sequence>
16. </xs:complexType>
17. </xs:schema>

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

First, you need to import that schema into this one so that you can see it. This is done by 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 the namespace "http://NamespaceTest.com/CustomerTypes".

So, to make use of the AddressType which is defined in CustomerTypes.xsd, and part of the namespace "http://NamespaceTest.com/CommonTypes", you must fully qualify it. To do this, you must define an alias for the namespace "http://NamespaceTest.com/CommonTypes". Again, this is done by using <xs:schema>.

The line *xmlns:cmn="http://NamespaceTest.com/CommonTypes"* specifies that the alias cmn represents the namespace "http://NamespaceTest.com/CommonTypes".

You now can make use of the types within the CommonTypes.xsd schema. When you do this, you must fully qualify them because they are not in the same targetNamespace as the schema that is using them. You do this as follows: *type="cmn:AddressType"*.

**OrderType.xsd**

1. <?xml version="1.0" encoding="utf-16"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <xs:schema **xmlns:cmn="http://NamespaceTest.com/CommonTypes"**
5. **targetNamespace="http://NamespaceTest.com/OrderTypes"**
6. xmlns:xs="http://www.w3.org/2001/XMLSchema"
7. elementFormDefault="qualified">
8. **<xs:import namespace="http://NamespaceTest.com/CommonTypes"**
9. **schemaLocation="CommonTypes.xsd" />**
10. <xs:complexType name="OrderType">
11. <xs:sequence>
12. <xs:element maxOccurs="unbounded" name="Item">
13. <xs:complexType>
14. <xs:sequence>
15. <xs:element name="ProductName" type="xs:string" />
16. <xs:element name="Quantity" type="xs:int" />
17. <xs:element name="UnitPrice" type="cmn:PriceType" />
18. </xs:sequence>
19. </xs:complexType>
20. </xs:element>
21. </xs:sequence>
22. </xs:complexType>
23. </xs:schema>

This schema defines the type OrderType that is within the namepace http://NamespaceTest.com/OrderTypes. The constructs used here are the same as those used in CustomerTypes.xsd.

**Main.xsd**

1. <?xml version="1.0" encoding="utf-16"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <xs:schema xmlns:ord="http://NamespaceTest.com/OrderTypes"
5. xmlns:pur="http://NamespaceTest.com/Purchase"
6. xmlns:cmn="http://NamespaceTest.com/CommonTypes"
7. xmlns:cust="http://NamespaceTest.com/CustomerTypes"
8. targetNamespace="http://NamespaceTest.com/Purchase"
9. xmlns:xs="http://www.w3.org/2001/XMLSchema"
10. elementFormDefault="qualified">
11. <xs:import schemaLocation="CommonTypes.xsd"
12. namespace="http://NamespaceTest.com/CommonTypes" />
13. <xs:import schemaLocation="CustomerTypes.xsd"
14. namespace="http://NamespaceTest.com/CustomerTypes" />
15. <xs:import schemaLocation="OrderTypes.xsd"
16. namespace="http://NamespaceTest.com/OrderTypes" />
17. <xs:element name="Purchase">
18. <xs:complexType>
19. <xs:sequence>
20. <xs:element name="OrderDetail" type="ord:OrderType" />
21. <xs:element name="PaymentMethod"
22. type="cmn:PaymentMethodType" />
23. <xs:element ref="pur:CustomerDetails"/>
24. </xs:sequence>
25. </xs:complexType>
26. </xs:element>
27. <xs:element name="CustomerDetails" type="cust:CustomerType"/>
28. </xs:schema>

The elements in this schema are part of the namespace "http://NamespaceTest.com/Purchase" (see the tagetNamespace attribute). This is your main schema and defines the concrete elements "Purchase" and "CustomerDetails".

This element builds on the other schemas, so you need to import them all and define aliases for each namesapce.

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

**The XML**

Becuase the root element Purchase is in the namespace "http://NamespaceTest.com/Purchase", you must quantify the <Purchase> element within the resulting XML document. Look at an example:

1. <?xml version="1.0"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <p:Purchase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation=
6. "http://NamespaceTest.com/Purchase Main.xsd"
7. xmlns:p="http://NamespaceTest.com/Purchase"
8. xmlns:o="http://NamespaceTest.com/OrderTypes"
9. xmlns:c="http://NamespaceTest.com/CustomerTypes"
10. xmlns:cmn="http://NamespaceTest.com/CommonTypes">
11. <p:OrderDetail>
12. <o:Item>
13. <o:ProductName>Widget</o:ProductName>
14. <o:Quantity>1</o:Quantity>
15. <o:UnitPrice>3.42</o:UnitPrice>
16. </o:Item>
17. </p:OrderDetail>
18. <p:PaymentMethod>VISA</p:PaymentMethod>
19. <p:CustomerDetails>
20. <c:Name>James</c:Name>
21. <c:DeliveryAddress>
22. <cmn:Line1>15 Some Road</cmn:Line1>
23. <cmn:Line2>SomeTown</cmn:Line2>
24. </c:DeliveryAddress>
25. <c:BillingAddress>
26. <cmn:Line1>15 Some Road</cmn:Line1>
27. <cmn:Line2>SomeTown</cmn:Line2>
28. </c:BillingAddress>
29. </p:CustomerDetails>
30. </p:Purchase>

The first thing you 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; a carriage return or space will do).

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 so it is part of the namespace "http://NamespaceTest.com/Purchase" even though it uses the complexType "OrderType" that 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 qualifiing within the namespace "http://NamespaceTest.com/OrderTypes".

**The Effect of elementFormDefault**

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

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

1. <?xml version="1.0"?>
2. <!-- Created with Liquid XML Studio 0.9.8.0
3. (http://www.liquid-technologies.com) -->
4. <p:Purchase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation=
6. "http://NamespaceTest.com/Purchase Main.xsd"
7. xmlns:p="http://NamespaceTest.com/Purchase">
8. <OrderDetail>
9. <Item>
10. <ProductName>Widget</ProductName>
11. <Quantity>1</Quantity>
12. <UnitPrice>3.42</UnitPrice>
13. </Item>
14. </OrderDetail>
15. <PaymentMethod>VISA</PaymentMethod>
16. <p:CustomerDetails>
17. <Name>James</Name>
18. <DeliveryAddress>
19. <Line1>15 Some Road</Line1>
20. <Line2>SomeTown</Line2>
21. </DeliveryAddress>
22. <BillingAddress>
23. <Line1>15 Some Road</Line1>
24. <Line2>SomeTown</Line2>
25. </BillingAddress>
26. </p:CustomerDetails>
27. </p:Purchase>

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

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

The first element is Purchase; this is defined gloablly in the Main.xsd schema, and therefore needs to be qualified within the schema's target namespace "http://NamespaceTest.com/Purchase".

The first child element is <OrderDetail> and is defined inline in Main.xsd->Purchase. 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 to be qualified with a namespace.

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

**Summary**

Namespaces provide a useful way of breaking schemas down into logical blocks, which can then be re-used throughout a company or project. The rules for namespacing in the resulting XML documents are rather complex; the rules provided here are a rough guide, things do get more complex as you dig further into it. For this reason tools to deal with these complexities are useful, see [XML Data Binding](http://www.liquid-technologies.com/Products_XMLDataBinding_ReasonsToUse.aspx).

This section covers a few of the lesser used constructs:

* Element and Attribute Groups
* any (Element)
* anyAttribute

**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> cannot 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.

1. <xs:group name="CustomerDataGroup">
2. <xs:sequence>
3. <xs:element name="Forename" type="xs:string" />
4. <xs:element name="Surname" type="xs:string" />
5. <xs:element name="Dob" type="xs:date" />
6. </xs:sequence>
7. </xs:group>
9. <xs:attributeGroup name="DobPropertiesGroup">
10. <xs:attribute name="Day" type="xs:string" />
11. <xs:attribute name="Month" type="xs:string" />
12. <xs:attribute name="Year" type="xs:integer" />
13. </xs:attributeGroup>

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

1. <xs:complexType name="Customer">
2. <xs:sequence>
3. **<xs:group ref="CustomerDataGroup"/>**
4. <xs:element name="..." type="..."/>
5. </xs:sequence>
6. **<xs:attributeGroup ref="DobPropertiesGroup"/>**
7. </xs:complexType>

**The <any> Element**

The <xs: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 you can still validate the message.

Look at the following schema:

1. <xs:element name="Message">
2. <xs:complexType>
3. <xs:sequence>
4. <xs:element name="DateSent" type="xs:date" />
5. <xs:element name="Sender" type="xs:string" />
6. <xs:element name="Content">
7. <xs:complexType>
8. <xs:sequence>
9. **<xs:any />**
10. </xs:sequence>
11. </xs:complexType>
12. </xs:element>
13. </xs:sequence>
14. </xs:complexType>
15. </xs:element>

You have defined an element called "Message" that 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.

1. <Message>
2. <DateSent>2000-01-12</DateSent>
3. <Sender>Admin</Sender>
4. <Content>
5. **<AccountCreationRequest>**
6. **<AccountName>Fred</AccountName>**
7. **</AccountCreationRequest>**
8. </Content>
9. </Message>

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

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, it must be valid against it, if there is no schema, that's Okay.
* ***Strict***: There must be a definition for the element available to the parser, and it must be valid against it.

**The <anyAttribute>**

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

1. <xs:element name="Sender">
2. <xs:complexType>
3. <xs:simpleContent>
4. <xs:extension base="xs:string">
5. **<xs:anyAttribute />**
6. </xs:extension>
7. </xs:simpleContent>
8. </xs:complexType>
9. </xs:element>

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

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