

XSD Tutorial - Part 1 of 5 - Elements and Attributes

Introduction

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 let's 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 etc. It can also describe the **values** that can be placed into any **element** or **attribute**.

A Note About Standards

- "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 2 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 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 XSD's ([See XML Studio, a FREE XSD development tool](#)), also when working with XML documents programmatically [XML Data Binding](#) is a much easier way to manipulate your documents (an 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 are using a design tool like [Liquid XML Studio](#).

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:

```
<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 [XSD standard for a complete list](#)). You can also create a user defined type using the `<xs:simple type>` and `<xs:complexType>` tags, but more on these later.

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 (failure to do this will cause a validation error). Examples of simple elements and their XML are below:

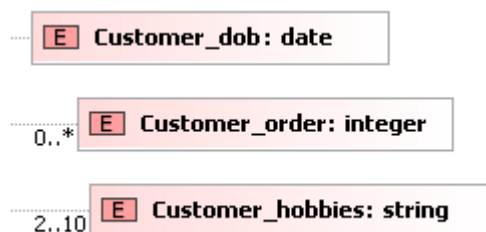
Sample XSD	Sample XML
<pre><xs:element name="Customer_dob" type="xs:date"/></pre>	<pre><Customer_dob> 2000-01-12T12:13:14Z</Customer_dob></pre>
<pre><xs:element name="Customer_address" type="xs:string"/></pre>	<pre><Customer_address> 99 London Road </Customer_address></pre>
<pre><xs:element name="OrderID" type="xs:int"/></pre>	<pre><OrderID> 5756 </OrderID></pre>

Cardinality

Specifying how many times an element can appear is referred to as cardinality, and is specified using the attributes `minOccurs` and `maxOccurs`. In this way, an element can be mandatory, optional, or appear many times. `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 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
<pre><xs:element name="Customer_dob" type="xs:date"/></pre>	If we don't specify <code>minOccurs</code> or <code>maxOccurs</code> , then the default values of 1 are used, so in this case there has to be one and only one occurrence of <code>Customer_dob</code>
<pre><xs:element name="Customer_order" type="xs:integer" minOccurs="0" maxOccurs="unbounded"/></pre>	Here, a <code>customer</code> can have any number of <code>Customer_orders</code> (even 0)
<pre><xs:element name="Customer_hobbies" type="xs:string" minOccurs="2" maxOccurs="10"/></pre>	In this example, the element <code>Customer_hobbies</code> must appear at least twice, but no more than 10 times



Simple Types

So far, we have touched on a few of the built in data types `xs:string`, `xs:integer`, `xs:date`. But you can also define your own types by modifying the 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 covered 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:

```
<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:

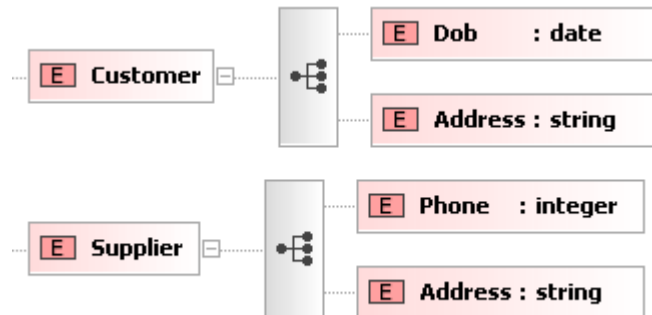
```
<xs:element name="Customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Dob" type="xs:date" />
      <xs:element name="Address" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Supplier">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Phone" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

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

```



Example XML

```

<Customer>
  <Dob> 2000-01-12T12:13:14Z </Dob>
  <Address> 34 thingy street, someplace, sometown, w1w8uu </Address>
</Customer>

<Supplier>
  <Phone>0123987654</Phone>
  <Address>22 whatever place, someplace, sometown, ssl 6gy </Address>
</Supplier>

```

What's changed?

Let's look at this in detail.

- 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**" do 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 2 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 English this is saying we can have an XML document that contains an element `<Customer>` which must have 2 child elements `<Dob>` and `<Address>`.

Compositors

There are 3 types of compositors `<xs:sequence>`, `<xs:choice>` and `<xs:all>`. These compositors allow us 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 compositors `<xs:sequence>` and `<xs:choice>` 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 as they are grouped into a single field. In the real world it would be better to break this out into a few fields. Let's fix this by breaking it out using the same technique shown above:

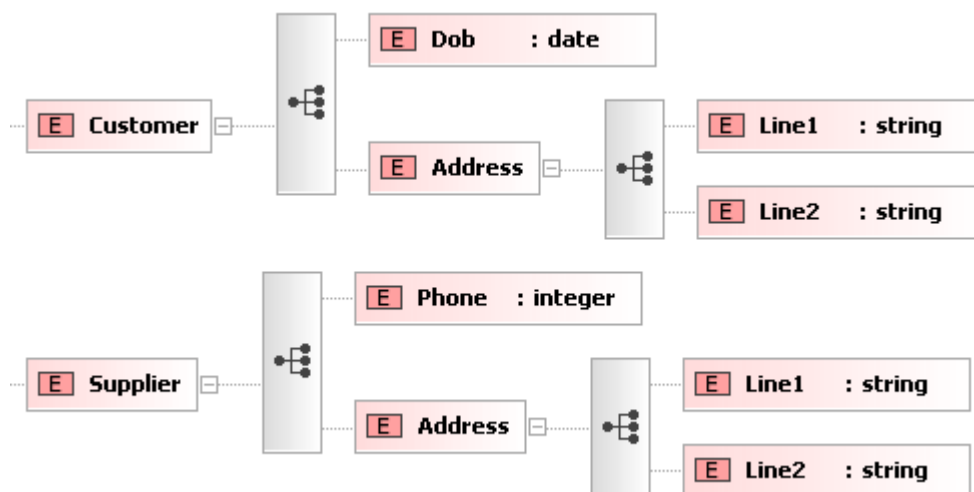
```
<xs:element name="Customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Dob" type="xs:date" />
      <xs:element name="Address">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Line1" type="xs:string" />
            <xs:element name="Line2" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

</xs:element>

<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 2 definitions for **Address**, which are the same.

Re-use

It would make much more sense to have 1 definition of "**Address**", that could 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">

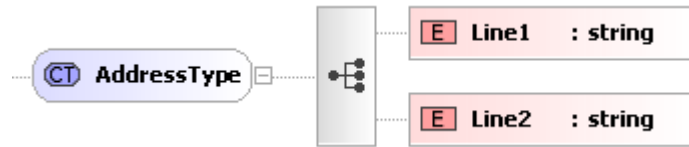
```



```

<xs:sequence>
  <xs:element name="Line1" type="xs:string"/>
  <xs:element name="Line2" type="xs:string"/>
</xs:sequence>
</xs:complexType>

```



We have now defined a `<xs:complexType>` that describes our representation of an `Address`, so let's use it.

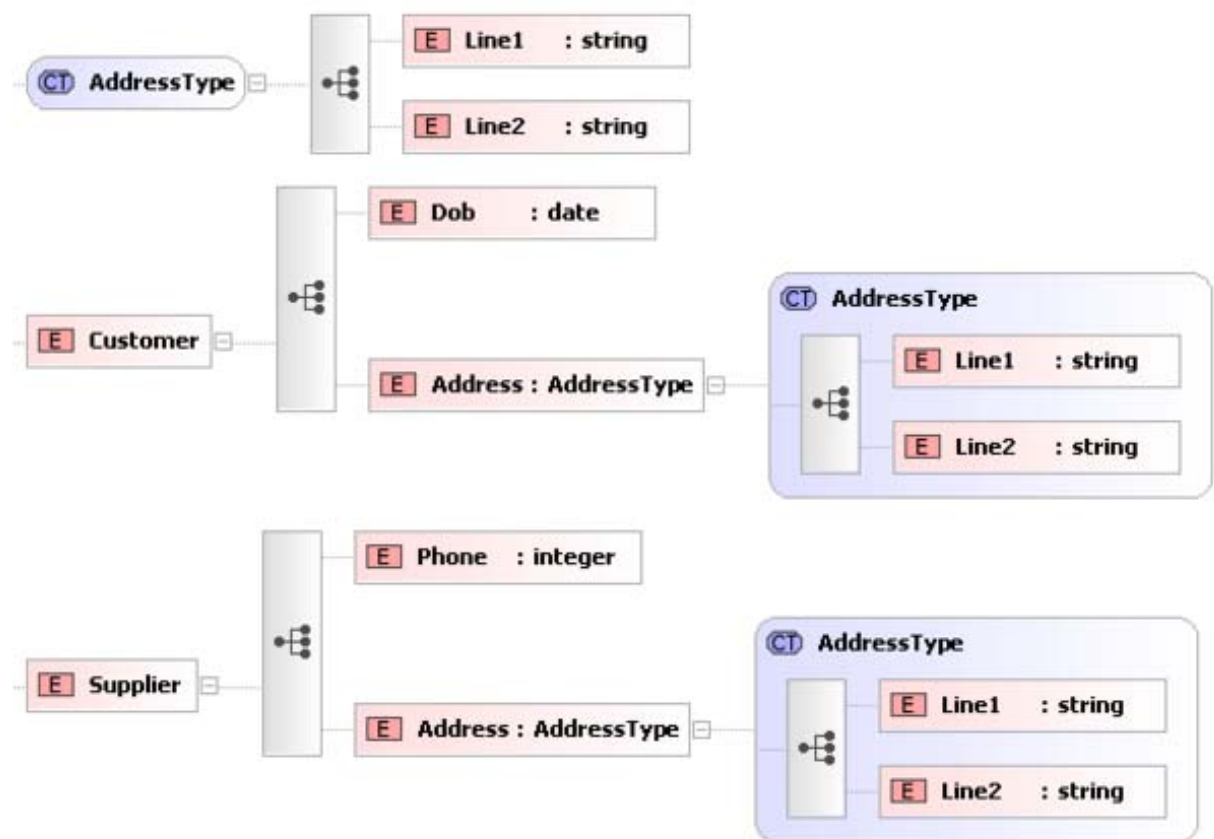
Remember when we started looking at elements and we said you could define your own type instead of using one of the standard ones (`xs:string`, `xs:integer`), well that's exactly what we were doing now.

```

<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="Phone" type="xs:integer"/>
      <xs:element name="Address" type="AddressType"/>
    </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**) we 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, 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 (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 cannot have a name (anonymous) and cannot be reused elsewhere.

Attributes

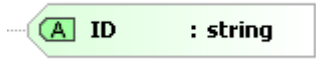
An `attribute` provides extra information within an element. `Attributes` are defined within an XSD as follows, having `name` and `type` properties.

```
<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"/>
```



To specify that an `attribute` must be present, use `<code>= "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`, this ties the `attribute` to the `element`. `Attributes` can also be specified globally and then referenced (but more about this later).

Sample XSD	Sample XML
<pre><xs:element name="Order"> <xs:complexType> <xs:attribute name="OrderID" type="xs:int"/> </xs:complexType> </xs:element></pre>	<pre><Order OrderID="6"/> or <Order/></pre>

<pre> <xs:element name="Order"> <xs:complexType> <xs:attribute name="OrderID" type="xs:int" use="optional"/> </xs:complexType> </xs:element> </pre>	<pre> <Order OrderID="6"/> or <Order/> </pre>
<pre> <xs:element name="Order"> <xs:complexType> <xs:attribute name="OrderID" type="xs:int" use="required"/> </xs:complexType> </xs:element> </pre>	<pre> <Order OrderID="6"/> </pre>

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 we have seen how an element can contain data, other **elements** or **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>

```

XSD Tutorial - Part 2 of 5 - Conventions & Recommendations

Introduction

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 meta data. Another way to look at it is that `attributes` are used for small pieces of data such as order id's, 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)

If 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 that 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, eg (`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 eg `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, eg `CustomerName`, should be `Name` within the sub `element Customer`.
- Only produce `complexType`s or `simpleTypes` for `types` that are likely to be re-used. If the structure only exists 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 XML standard)
 - `xmlns` (defined in Namespaces in XML standard)
 - `xs` <http://www.w3.org/2001/XMLSchema>
 - `xsi` <http://www.w3.org/2001/XMLSchema-instance>
- Try to think about versioning early on in your schema design. If it is 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. See [Versioning recommendations](#).
- 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 namespaces in the resulting XML simpler (if not more verbose).

XSD Tutorial - Part 3 of 5 - Extending Existing Types

Introduction

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, we would call this inheritance or sub classing. The same concepts also exist in the XSD standard. This allows us to take an existing `type` definition and extend it. We can also 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. Lets see how this may be useful with an example.

Looking at the `AddressType` that we defined earlier ([in part 1](#)), 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="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">
```

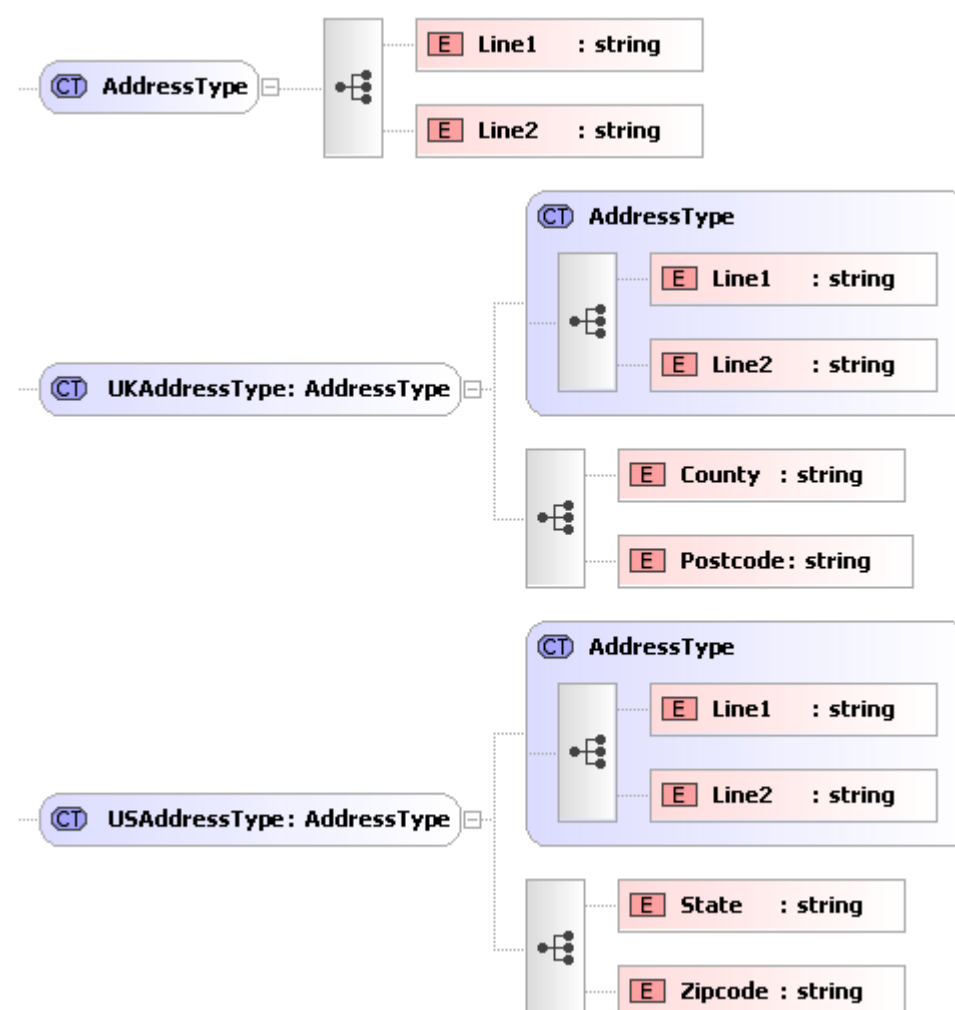
```

<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>

```

This is clearer when viewed graphically. But basically it is saying - 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".

There are 2 new things here the `<xs:extension>` element and the `<xs:complexContent>` element; we'll get to these shortly.



We can now use these new types as follows:


```
<xs:element name="UKAddress" type="UKAddressType" />
<xs:element name="USAddress" type="USAddressType" />
```

Some sample XML for these **elements** may look like this:

```
<UKAddress>
  <Line1>34 thingy street</Line1>
  <Line2>someplace</Line2>
  <County>somerset</County>
```

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.

Note: Because we are restricting an existing **type**, the only definitions that can appear in the **<xs:restriction>** are a subset 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" />
```

Some Sample XML for this element may look like this.

```
<InternalAddressType>
  <Line1>Desk 4, Second Floor</Line1>
</InternalAddressType>
```

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

Use of Extended/Restricted Types

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 specifies it is of **type** **UKAddressType**, then that is what must appear in the XML document. But if an element specifies it is of **type** "**AddressType**", then any of the 4 **types** can appear in the XML

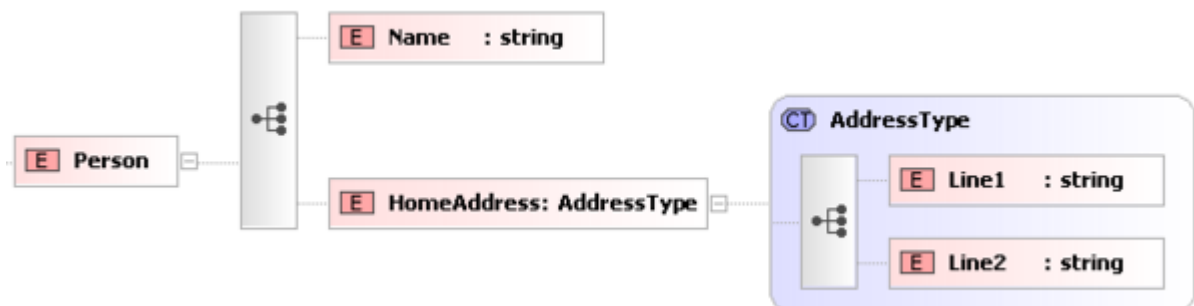
document (`UKAddressType`, `USAddressType`, `InternalAddressType` or `AddressType`).

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 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`) then you have to let the parser know which `type` you are using. This is done in the XML document using the `xsi:type` attribute.

Lets 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>
```



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

```
<?xml version="1.0"?>
<Person>
  <Name>Fred</Name>
  <HomeAddress>
    <Line1>22 whatever place, someplace</Line1>
    <Line2>sometown, ss1 6gy </Line2>
  </HomeAddress>
</Person>
```

But 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>

```

Lets 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 name space `"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"` 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.

But more about namespaces in the next section.

Extending Simple Types

There are 3 ways in which a `simpleType` can be extended; `Restriction`, `List` or `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"`, and 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>

```

Shown graphically in [Liquid XML Studio](#) as follows



Content	
Name	LetterType
Simple Type Content	Restriction
Type	 string
Facets	
Enumerations	String[] Array
Length	
Max Length	
Min Length	
Pattern	[a-zA-Z]
Properties	
Final	None
Id	

Let's go through this line by line:

1. A **<simpleType>** tag is used to define 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**". For a complete list, see the [XSD Standard](#), 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.	<pre> <xs:minLength value="3"> <xs:maxLength value="8"> </pre>	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	<pre> <xs:minInclusive value="0"><xs:maxInclusive value="10"> </pre>	The value must be between 0 and 10

Overview	Syntax	Syntax explained
<p>The lower and upper range for numerical values</p> <p>The value must be less than or greater than</p>	<pre><xs:minExclusive value="0"><xs:maxExclusive value="10"></pre>	<p>The value must be greater than 0 and less than 10</p>
<p>The exact number of characters allowed</p>	<pre><xs:length value="30"><code></pre>	<p>The length must be exactly 30 characters</p>
<p>Exact number of digits allowed</p>	<pre><xs:totalDigits value="9"></pre>	<p>Can not have more than 9 digits</p>
<p>A list of values allowed</p>	<pre><xs:enumeration value="Hippo"/> <xs:enumeration value="Zebra"/> <xs:enumeration value="Lion"/></pre>	<p>The only permitted values are Hippo, Zebra or Lion</p>
<p>The number of decimal places allowed (must be ≥ 0)</p>	<pre><xs:fractionDigits value="2"/></pre>	<p>The value has to have 2 decimal places</p>
<p>This defines how whitespace will be handled. Whitespace is line feeds, carriage returns, tabs, spaces, etc.</p>	<pre><xs:whitespace value="preserve"/> <xs:whitespace value="replace"/> <xs:whitespace value="collapse"/></pre>	<p>Preserve - Keeps all whitespaces Replace - Replaces all whitespace with a single space Collapse - Replaces all whitespace characters with a single space, then if there are multiple spaces together then they will be reduced to one space.</p>
<p>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</p>	<pre><xs:pattern value="[0-999]"/></pre>	<p>[0-999] - 1 digit only between 0 and 999 [0-99][0-99][0-99] - 3 digits all have to be between 0 and 999 [a-z][0-10][A-Z] - 1 st digit has to be between a and z</p>

Overview	Syntax	Syntax explained
		<p>and 2nd digit has to be between 0 and 9, the 3rd digit has to be between 10 and the 3rd digit is between A and Z. These are case sensitive.</p> <p>[a-zA-Z] - 1 digit that can be either lower or uppercase A – Z</p> <p>[123] - 1 digit that has to be 1, 2 or 3</p> <p>([a-z])* - Zero or more occurrences of a to z</p> <p>([q][u])+ - Looking for a pair letters that satisfy the criteria, in this case a q followed by a u</p> <p>([a-z][0-999])+ - As above, looking for a pair where the 1st digit is lowercase and between a and z, and the 2nd digit is between 0 and 999 for example a1, c99, z999, f45</p> <p>[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.</p>

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](#).

Union

A **union** is a mechanism for combining 2 or more different data **types** into one.

The following defines 2 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>
```


ST SizeByNumberType: positiveInteger

ST SizeByStringNameType: string

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>
```

ST USClothingSizeType: <Simple Type Union>

Content	
Name	USClothingSizeType
Simple Type Content	Union
Type	 <Simple Type Union>
UnionTypes	SizeByNumberType, SizeByStringNameType
Properties	
Final	None
Id	


This means the **type** can contain any of the values that the 2 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>
```

 SizesinStockType: SizeByNumberType

Content	
Name	SizesinStockType
Simple Type Content	List
Type	 SizeByNumberType
Facets	
Enumerations	String[] Array
Fractional Digits	
Max Exclusive	
Max Inclusive	
Min Exclusive	
Min Inclusive	
Pattern	
Total Digits	
Properties	
Final	None
Id	

XSD Tutorial - Part 4 of 5 - Namespaces

Namespaces

So far we have glossed over namespaces entirely, we will hopefully address this a little now. Firstly the full namespacing rules are rather complicated, so this will just be an overview. If you are working with a schema that makes use of namespaces then [XML Data Binding](#) will save you a great deal of time as it takes this complexity away. If you're not using a data binding tool then you may want to refer to the [XSD standard](#) or purchase a book!

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.

In this example, the schema is broken out into 4 files.

- *CommonTypes* - this could contain all your basic **types**, **AddressType**, **PriceType**, **PaymentMethodType** etc.
- *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.

All this 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, i.e.:

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

```
</xs:schema>
```

The value of `targetNamespace` is just a unique identifier, typically companies use their 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 as the `targetNamespace` and so some XML parsers will use this as a hint path for the schema, e.g.:

`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 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"?>
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com) -->
<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>

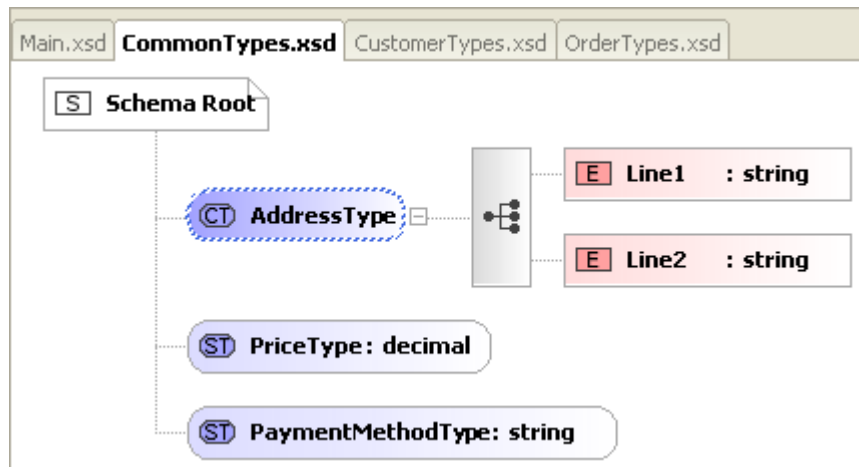
  <xs:simpleType name="PriceType">
    <xs:restriction base="xs:decimal">
      <xs:fractionDigits value="2" />
    </xs:restriction>
  </xs:simpleType>

  <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>
```

```

</xs:simpleType>
</xs:schema>

```



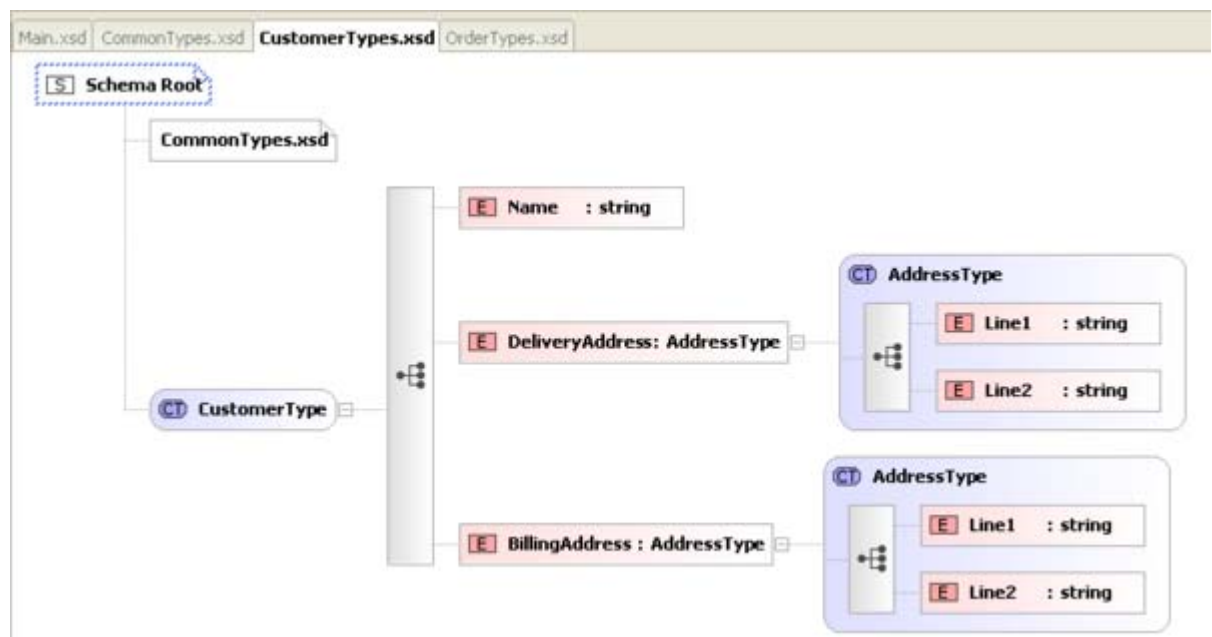
This schema defines some basic re-usable entities and **types**. The use of the **targetNamespace** attribute in the **<xs:schema>** element ensures all the enclosed definitions (**AddressType**, **PriceType** and **PaymentMethodType**) are in the namespace "**http://NamespaceTest.com/CommonTypes**".

CustomerTypes.xsd

```

<?xml version="1.0" encoding="utf-16"?>
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com) -->
<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 the namespace "*http://NamespaceTest.com/CustomerTypes*".

So in order to make use of the **AddressType** which is defined in *CustomerTypes.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*". Again this is done using `<xs:schema>`.

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

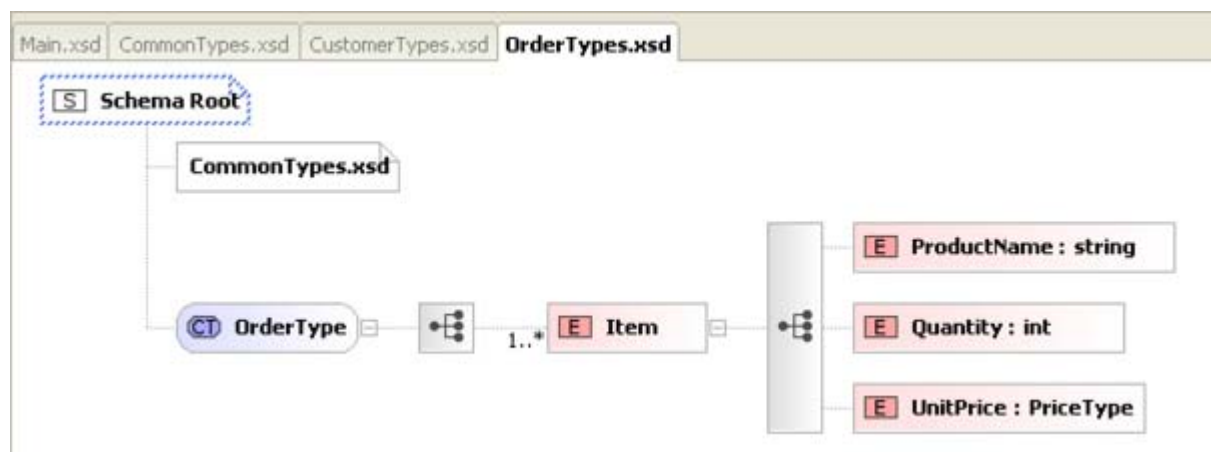
We can now make use of the types within the *CommonTypes.xsd* schema. When we do this we must fully qualify them as they are not in the same **targetNamespace** as the schema that is using them. We do this as follows:
`type="cmn:AddressType"`.

OrderType.xsd

```

<?xml version="1.0" encoding="utf-16"?>
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com)
-->
<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 namespace="http://NamespaceTest.com/CommonTypes"
              schemaLocation="CommonTypes.xsd" />
    <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`.

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

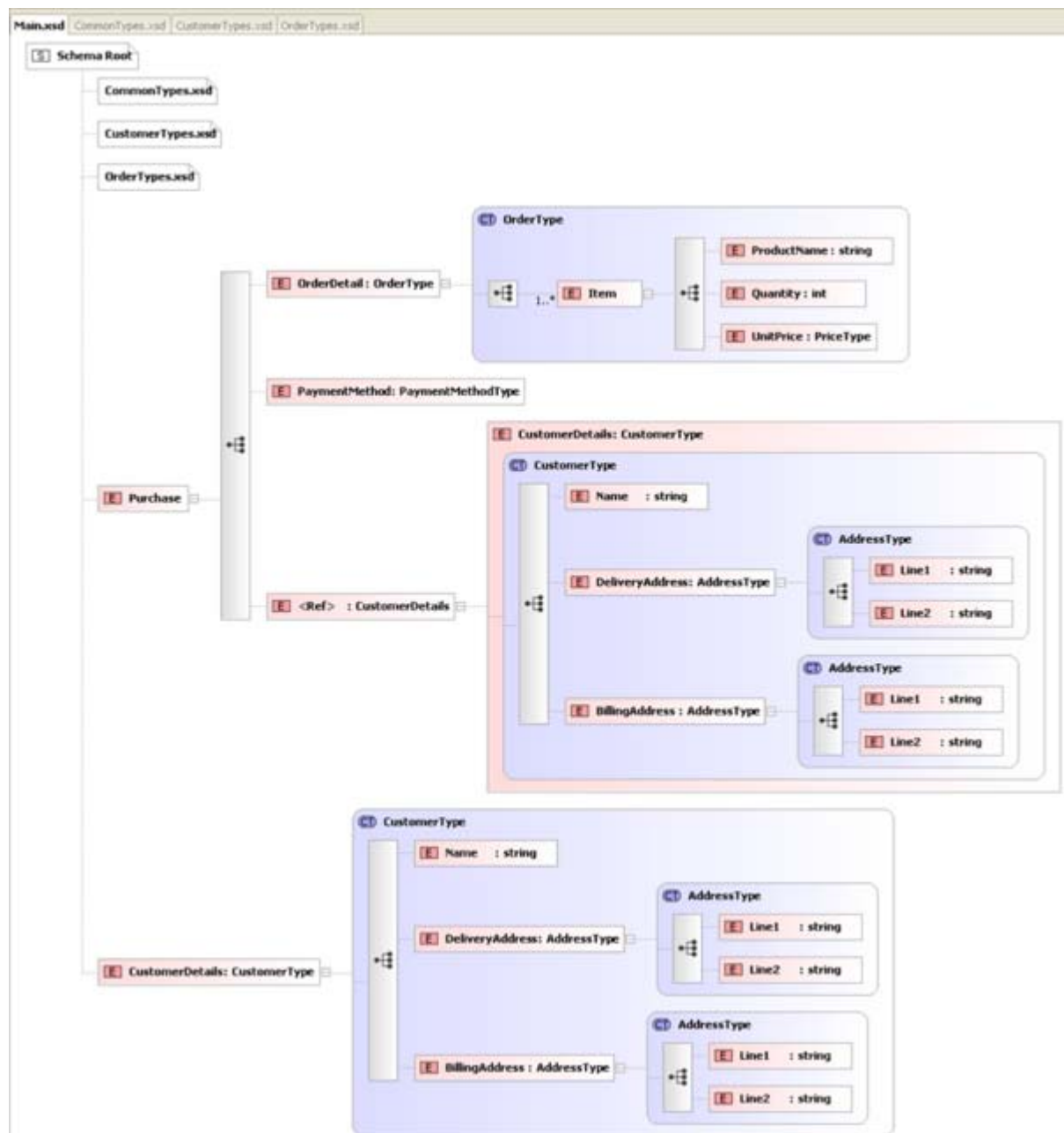
Main.xsd

```

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

```

```
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com)
-->
<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`" (see `targetNamespace` attribute).

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`".

The XML

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

```
<?xml version="1.0"?>
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com)
-->
<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 the 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 - 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 that 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 Effect of elementFormDefault

You may have noticed that each schema contained an **attribute** `elementFormDefault="qualified"`. This has 2 possible values, `qualified`, and `unqualified`, the **default** is `unqualified`. This **attribute** changes the namespacing 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"?>
<!-- Created with Liquid XML Studio 0.9.8.0 (http://www.liquid-technologies.com)
-->
<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>**.

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](#).

XSD Tutorial - Part 5 of 5 - Other Useful Bits

Introduction

This section covers a few of the lesser used constructs:

- `Element` and `Attribute` Groups
- The `<any>` Element
- `<anyAttribute>`

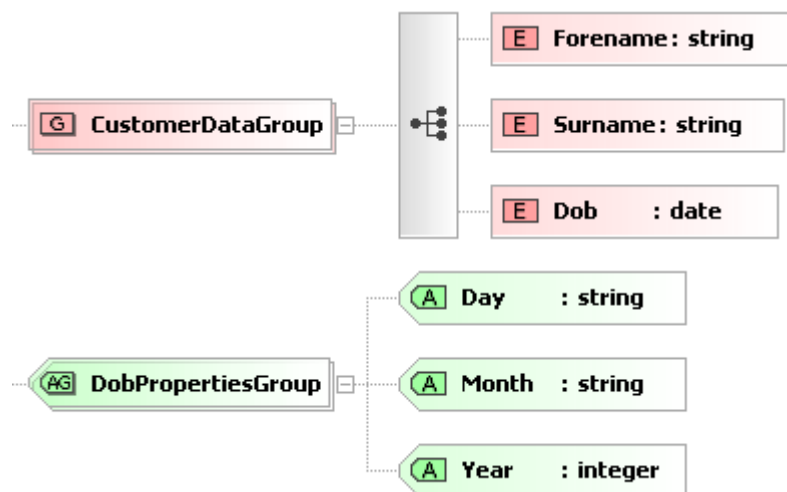
Element and Attribute Groups

`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.

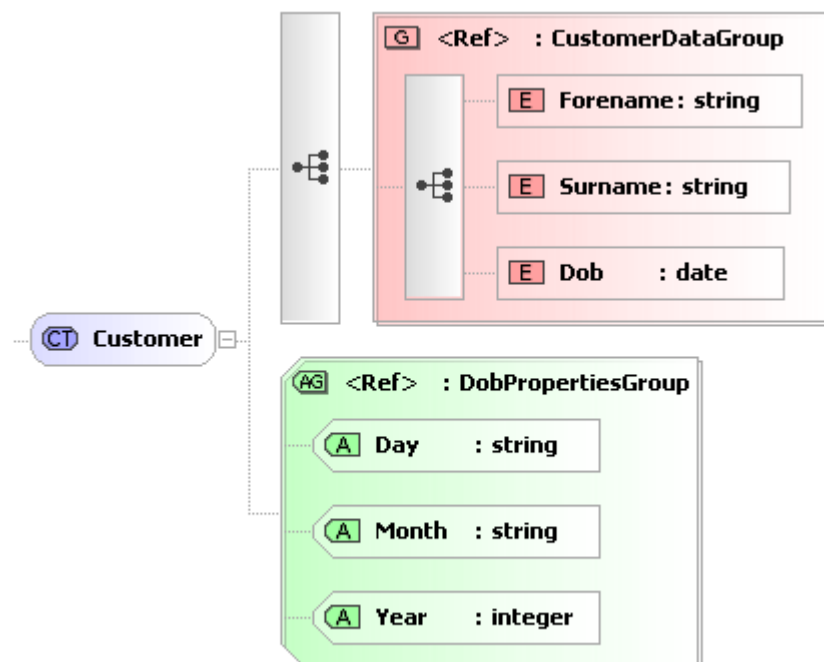
```
<xs:group name="CustomerDataGroup">
  <xs:sequence>
    <xs:element name="Forename" type="xs:string" />
    <xs:element name="Surname" type="xs:string" />
    <xs:element name="Dob" type="xs:date" />
  </xs:sequence>
</xs:group>

<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>
```



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 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 `<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 many instances of undefined `elements` must be placed within the XML document.

Namespace allows you to specify that the undefined **element** "must" belong to the given namespace. This may be a list of namespace's (space separated). There are also 3 built in values **##any**, **##other**, **##targetnamespace**, **##local**. Consult the XSD standard 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.

<anyAttribute>

<xs:anyAttribute> works in exactly the same way as **<xs: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>
```