Extending Existing Types

This article gives an overview of some of the more advanced topics of XML Schemas and how to use them.

* [Overview](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-extending-types#overview)
* [Extending Complex Types](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-extending-types#extending-complex-types)
* [Restricting Complex Types](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-extending-types#restricting-complex-types)
* [Using the xsi:type Attribute](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-extending-types#xsi-type)
* [Extending Simple Types](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-extending-types#extending-simple-types)

**Data Types** Overview

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

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

**Extending** Complex Types

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

Looking at the AddressType that we defined earlier (in [Part 1](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes)), let's assume our company has now gone international and we need to capture country specific addresses. In this case we need specific information for UK addresses (County and Postcode), and for US addresses (State and ZipCode).

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

<xs:complexType name="AddressType">

<xs:sequence>

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

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

</xs:sequence>

</xs:complexType>

<xs:complexType name="UKAddressType">

<xs:complexContent>

<xs:extension base="AddressType">

<xs:sequence>

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

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

</xs:sequence>

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:complexType name="USAddressType">

<xs:complexContent>

<xs:extension base="AddressType">

<xs:sequence>

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

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

</xs:sequence>

</xs:extension>

</xs:complexContent>

</xs:complexType>

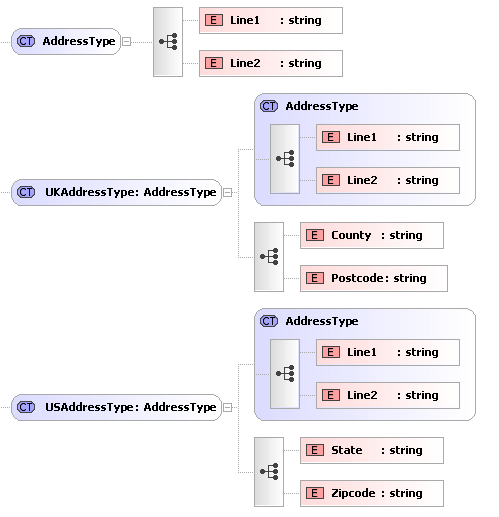
Notice each of the two new address types extend the original 'base' address type using:

<xs:extension base="AddressType">

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

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

This is clearer when viewed graphically:

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

We can now use these new types as follows:

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

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

Sample XML for these elements may look like this:

<UKAddress>

<Line1>34 thingy street</Line1>

<Line2>someplace</Line2>

<County>somerset/County>

<Postcode>w1w8uu</Postcode>

</UKAddress>

<USAddress>

<Line1>234 Lancaster Av</Line1>

<Line2>Smallville</Line2>

<State>Florida</State>

<Zipcode>34543</Zipcode>

</USAddress>

Learn faster with Liquid XML **Free Community Edition** [Download Now!](https://www.liquid-technologies.com/trial-download)

**Restricting** Complex Types

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

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

<xs:complexType name="AddressType">

<xs:sequence>

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

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

</xs:sequence>

</xs:complexType>

<xs:complexType name="InternalAddressType">

<xs:complexContent>

<xs:restriction base="AddressType">

<xs:sequence>

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

</xs:sequence>

</xs:restriction>

</xs:complexContent>

</xs:complexType>

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

<xs:restriction base="AddressType">

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

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

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

We can now use this new type as follows:

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

Sample XML for this element may look like this:

<InternalAddressType>

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

</InternalAddressType>

Learn faster with Liquid XML Free Community Edition [Download Now!](https://www.liquid-technologies.com/trial-download)

Using the **xsi:type** Attribute

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

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

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

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

Let's look at an example:

<xs:element name="Person">

<xs:complexType>

<xs:sequence>

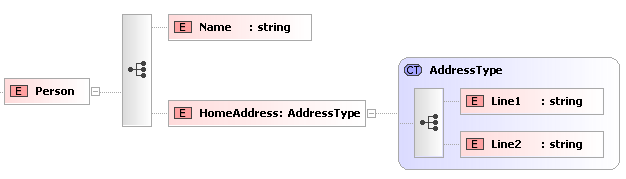
<xs:element name="Name" type="xs:string" />

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

</xs:sequence>

</xs:complexType>

</xs:element>

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

Sample XML for the above may look like the following:

<?xml version="1.0" ?>

<Person>

<Name>Fred</Name>

<HomeAddress>

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

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

</HomeAddress>

</Person>

However, the following is also valid:

<?xml version="1.0" ?>

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

<Name>Fred</Name>

<HomeAddress xsi:type="USAddressType">

<Line1>234 Lancaseter Av</Line1>

<Line2>SmallsVille</Line2>

<State>Florida</State>

<Zipcode>34543</Zipcode>

</HomeAddress>

</Person>

Let's look at that in more detail.

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

We'll learn more about [namespaces](https://www.liquid-technologies.com/xml-schema-tutorial/xsd-namespaces) in the next section.

Try Liquid Studio and see how we can help you today [Free Community Edition](https://www.liquid-technologies.com/trial-download)

**Extending** Simple Types

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

Restriction

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

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

<xs:simpleType name="LetterType">

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

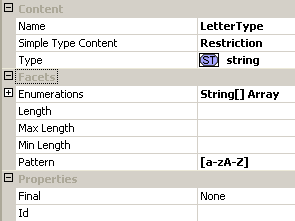
<xs:pattern value="[a-zA-Z]" />

</xs:restriction>

</xs:simpleType>

This can be shown graphically in Liquid Studio as follows:

[Simple Type Content](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-type-content.png)

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

Let's go through this line by line.

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

Restrictions may also be referred to as **Facets**. For a complete list see the [W3C XSD Standard](http://www.w3.org/TR/xmlschema-2/#rf-facets), but to give you an idea, here are a few examples:

| **Overview** | **Syntax** | **Syntax explained** |
| --- | --- | --- |
| The minimum and maximum length allowed. | <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. | <xs:minInclusive value="0">  <xs:maxInclusive value="10"> | The value must be between 0 and 10. |
| The lower and upper range for numerical values. | <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 be exactly 30 characters. |
| The maximum number of digits allowed. | <xs:totalDigits value="9"> | The value must have no more than 9 digits. |
| The maximum number of decimal places allowed. | <xs:fractionDigits value="2"> | The value must have no more than 2 decimal places. |
| 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. |
| This defines how whitespace will be handled (e.g. line feeds, carriage returns, tabs, spaces). | <xs:whitespace value="preserve">  <xs:whitespace value="replace">  <xs:whitespace value="collapse"> | Preserve - Keeps all whitespace. Replace - Replaces each whitespace with a space. Collapse - Replaces each whitespace character with a space and then reduces multiple spaces to one space. |
| Defines the character pattern allowed using regular expressions. For a complete list see the W3C XSD Standard: [Regular Expressions](http://www.w3.org/TR/xmlschema-2/#regexs). | <xs:pattern value="[0-9]"> | [0-9] - 1 digit only between 0 and 9.  [0-9][0-9][0-9] - 3 digits all have to be between 0 and 9.  [a-z][0-9][A-Z] - 1st digit has to be between a and z and 2nd digit has to be between 0 and 9 and the 3rd digit is between A and Z. These are case sensitive.  [a-zA-Z] - 1 digit that can be either lower or upper case A to Z.  [123] - 1 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 letters that satisfy the criteria, in this case a q followed by a u.  ([a-z][0-9])+ - As above, looking for a pair where the 1st digit is lower case and between a and z, and the 2nd digit is between 0 and 9, for example a1, c2, z159, f45.  [a-z0-9]{8} - Must be exactly 8 characters in a row and they must be lower case 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 W3C XSD standard.*

Union

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

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

<xs:simpleType name="SizeByNumberType">

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

<xs:maxInclusive value="21" />

</xs:restriction>

</xs:simpleType>

<xs:simpleType name="SizeByStringNameType">

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

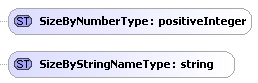
<xs:enumeration value="small" />

<xs:enumeration value="medium" />

<xs:enumeration value="large" />

</xs:restriction>

</xs:simpleType>

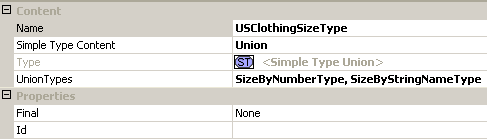
[](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-types.png)

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>

[Simple Type Union](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-type-union.png) [](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-type-union-properties.png)

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

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

List

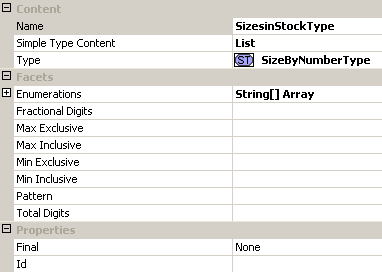
A list allows the value (in the XML document) to contain a number of valid values separated by whitespace.

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

<xs:simpleType name="SizesinStockType">

<xs:list itemType="SizeByNumberType" />

</xs:simpleType>

[Simple Type List](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-type-list.png) [](https://www.liquid-technologies.com/Content/images/tutorials/xsd-tutorial/simple-type-list-properties.png)