Birkbeck
UNIVERSITY OF LONDON

XML

Session 5

XML Schemas

Doing more with XML Schemas

- XML Schemas allow a much more fine tuned declaration of models than DTDs.
- <element name="firstname" type="string" /> describes an XML element with simple content, a container for a simple value, in this case a text string. Using different types you can specify what kind of text the element can hold.
  - DTD equivalent <!ELEMENT firstname (#PCDATA)>
- On the other hand, declarations such as <element name="firstname" /> allow any kind of content to be added inside the firstname element.
  - DTD equivalent: <!ELEMENT firstname ANY>
  - <firstname><emphasis>J</emphasis>ohn</firstname> is valid

Complex content in XML Schemas

- In order to declare an element that will have more than just a simple value the **complexType** element is used.
- Complex types enable the declaration of child elements and attributes for an element.

```
<element name="employee">
   <complexType>
     <!-- … -->
   </complexType>
</element>
```

Empty element in XML Schemas

- Some elements can act like "placeholders" with no content and no attributes, e.g. a pageBreak element which forces page breaks to occur at earlier and more convenient points than they otherwise would.
- An empty complexType element declares an empty element in XML Schemas:

```
<element name="pageBreak">
   <complexType></complexType>
</element>
```

- Note the model does not distinguish between <pageBreak /> and <pageBreak></pageBreak>. Both are legal representations of an empty element in XML.

Child element content

- Elements that need to contain other elements as children can do so by referring to elements that have already been defined elsewhere. Each reference requires anther Element element, but this time holding a **ref** attribute (instead of a name attribute) to refer to the declaration concerned.
- These references cannot be placed directly within the complexType element. They must be wrapped in another element that indicates how they are to be combined with each other. If the embedded elements must occur in a pre-defined order then the **sequence** element is used, otherwise the **choice** element can be used.
- The use of namespaces is important when using references.

Group occurrence options

- Just as it is possible to specify that a particular element is optional, required or repeatable (with minOccurs and maxOccurs attributes), it is also possible to make a sequence or choice group optional and repeatable.
- The same two occurrence attributes minOccurs and maxOccurs can also be used on the sequence and choice elements. A minOccurs of 0 makes all the elements in a group optional.

```
<xs:sequence minOccurs="0">
   <xs:element ref="firstname" />
   <xs:element ref="lastname" />
</xs:sequence>
```

Group occurrence options

- Similarly, when maxOccurs is given a value greater than one, an entire sequence group can be repeated:

```
<xs:sequence maxOccurs="unbounded">
  <xs:element ref="firstname" />
  <xs:element ref="lastname" />
</xs:sequence>
```

- Valid XML with respect to the schema:

```
<firstname>…</firstname><lastname>…</lastname><firstname>…
</firstname><lastname>…</lastname>
```

Hands on exercise 12

- Now do hands on exercise 12

Embedded groups

- The **choice** and **sequence** elements can contain each other.
  - There are no limitations on either the depth these models are nested within each other, or on the number of occurrences of one or both model types within another.

```
<xs:sequence>
   <xs:element> ref="title" />
   <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="para" />
      <xs:element ref="list" />
      <xs:element ref="table" />
   </xs:choice>
</xs:sequence>
```

Mixed content

- To allow text to appear in the content model, the **mixed** attribute is given the value true (the default value is false). Typically, mixed content involves the unconstrained ordering of inter-mixed sub-elements, where the entire group is generally made optional and repeatable.

```
<!-- XML to validate -->
<para>Are <emph>you</emph> going to
<name>Scarborough</name> fair?</para>
<!-- DTD declaration -->
<!ELEMENT para (#PCDATA | emph | name)*>
```

Mixed content

```
<!-- XML Schema declaration -->
<xs:element name="para">
   <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="emph" />
         <xs:element ref="name" />
      </xs:choice>
   </xs:complexType>
</xs:element>
```

- Unlike DTDs it is not necessary for the group to be an option group and it is also not ncecessary to make the group optional and repeatable. The text around sub-elements is always optional, regardless of the minOccurs attribute value on the group.

Mixed content

```
<!-- XML Schema declaration -->
<xs:element name="para">
   <xs:complexType mixed="true">
      <xs:choice> <!-- required and not repeatable -->
         <xs:element ref="name" /><!-- required element -->
      </xs:choice>
   </xs:complexType>
</xs:element>
<!-- Valid XML -->
<para><name>John Smith</name></para>
<para>The famous <name>John Smith</name></para>
<!-- Invalid XML -->
<para>John Smith</para> <!-- ERROR -->
```

Attributes

- Attributes are created and referenced in essentially the same way as elements. The **attribute** element is used to create an attribute.

```
<xs:attribute name="security" type="xs:string" />
```

- The type attribute is used to specify the type of value that can be contained. The type must always be a simple data type and never a complex type, because attributes cannot contain either element tags or sub-attributes.
- The attribute declaration appears at the end of a complex type definition.

Attribute example

```
<element name="chapter">
  <complexType>
    …
    <attribute name="security" type="string" />
  </complexType>
</element>

<!-- DTD equivalent -->
<!ATTLIST chapter security CDATA #IMPLIED>
```

Attributes on simple elements

- When an element has simple content, it does not normally require the complexType element to help define it. A simple technique to declare an attribute is to use the mixed content attribute.

```
<element name="para">
   <complexType mixed="true">
     <attribute name="security" type="string" />
   </complexType>
</element>
```

Required attributes

- By default, attributes are optional. They do not have to be present in every element instance, or indeed in any instance. To make the attribute required, add the **use** attribute with a value of **required**.

```
<attribute name="security" type="string" use="required" />
```

Default values for attributes

- When an attribute is declared it can be given a default value. The value specified within the **default** attribute is applied when the attribute is not present on the element.
- The use attribute must have a value of optional, or simply be absent, for this to work.

```
<attribute name=“security” type=“string” default=“secret” />
```

Fixed values for attributes

- It is possible to specify a value for the attribute that can never be changed. Use the **fixed** attribute value to provide the fixed value.

```
<attribute name="security" type="string" fixed="secret" />
```

- Because the value is always present or implied, a use attribute value of 'optional' does not apply, and neither does the default attribute.
- Note that the default and fixed values can be applied to element content as well as to attribute values.

Attribute groups

- When a number of element types require the same set of attributes, it is possible to pre-define and name a group of attributes, then refer to this group in each element definition. The **attributeGroup** element has the usual **name** and **ref** attributes, and contains any number of attribute definitions and attribute references.
- When used as a reference, it appears at the end of a complexType declaration.

Attribute groups example

```
<xs:attribute name="security" … />
<xs:attributeGroup name="standardAtts">
   <xs:attribute name="id" type="ID" use="required" />
   <xs:attribute name="indentLevel" type="xs:string" />
   <xs:attribute ref="security" />
</xs:attributeGroup>
<xs:element name="chapter">
   <xs:complexType>
      …
      <xs:attributeGroup ref="standardAtts" />
   </xs:complexType>
</xs:element>
```

Hands on exercise 13

- Now do hands on exercise 13

All groups

- Choice and sequence elements can be inadequate to accurately model a required structure. Consider the need for a book cover to contain title, author name and publisher name, but for the order of these three items to be unconstrained, so that a document instance author can decide which should appear first and last.
- The sequence element would be too restrictive, and the option element, which would need to be allowed to repeat, would not be restrictive enough (it would, for example, allow two titles to appear).

All groups

- What is needed is a construct that specified a list of required elements, but does not dictate the order in which they should occur
- The **all** element is designed for this purpose:

```
<xs:complexType>
   <xs:all>
      <xs:element ref="bookTitle" />
      <xs:element ref="authorName" />
      <xs:element ref="publisher" />
   </xs:all>
</xs:complexType>
```

All groups limitations

- A group of this kind cannot contain text. However, the elements in the group may do so, and the `mixed` attribute can still be used on the `complexType` element.
- When in a mixed content model, the containing element can contain text before, between and after the sequence of elements. If the group is also made optional, then it is possible that only text will occur.
- Unlike the other grouping types, this group type cannot contain embedded groups of any type. It can only contain element definitions and element references.
- The other group types cannot contain this kind of group. It must be the top and only level of a content model.

All group mixed model example

```
<xs:complexType mixed="true">
   <xs:all minOccurs="0" maxOccurs="1">
      <xs:element ref="bookTitle" />
      <xs:element ref="authorName" />
      <xs:element ref="publisher" />
   </xs:all>
</xs:complexType>
```

- Validates:

```
<cover>This book cover is not defined yet</cover>
<cover>
   Title: <bookTitle>…</bookTitle>
   Author: <authorName>…</authorName>
   Publisher: <publisher>…</publisher>
</cover>
```

Simple types with attributes

- When an element has simple content, but also has attributes, it cannot simply refer to a data type, such as 'string'. Instead it must include the complexType element, so that attribute declarations can be made. However, there remains the problem of how to specify the simple data type of the element content.
- The **simpleContent** element is used to indicate that the complex element will actually be based on a simple data type.
- The **extension** element is employed, both to specify the datatype of the element in its **base** attribute, and to extend the type by adding the required attribute.

Simple types with attributes example

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

Local element definitions

- It is not necessary to define an element in one place, then refer to this definition in the content model of another element definition. When the element concerned is only used within the model of one other element, it is more efficient to simply define the sub-element in-situ within the model of the outer element.
- The embedded element definition uses the name attribute instead of the ref attribute.
- Sub-elements of these inserted definitions can also be defined locally.
- Local elements have no existence beyond the location where they are defined. They cannot be referenced from other content models.

Local element definition example

```
<xs:element name="book">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="title" />
         <xs:element name="chapter" maxOccurs="unbounded">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="title" />
                  <xs:element name="para" type="xs:string" />
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Redefining elements

- Using local definitions, it is possible for the schema to include several, context-specific element definitions that have the same name, but different content models or attributes.
- A local definition overrides a global definition with the same name, and is also clearly distinct from any other definitions that are local to other element models.

Redefining elements example

```
<xs:element name="title" type="xs:string" />
<xs:element name="chapter">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="title">...</xs:element>
         ...
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="person">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="title">...</xs:element>
         ...
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Simple type derivations

- One of the strengths of XML Schema is the ability it gives schema authors to create new simple data types that are specific to the document model being defined.
- A new data type is *derived* from an existing type by first **referencing** an existing simple data type, then creating a new one that is either a *restricted* version or an *extended* version of that type.
- Once created the new type can be referenced in the usual way from element and attribute definitions.
- For example, it is possible to create a new numeric type that is based on the 'integer' type, but is constrained to a value between 5 and 50.
- `<xs:element name="score" type="<b>scoreRange</b>" />`

Simple type derivations

- New simple data types are created using the **simpleType** element and named using the **name** attribute.

```
<simpleType name="scoreRange">…</simpleType>
```

- This element can either
  - create restricted version of an existing data type.
  - define a list type
  - create a new type that is a combination of two or more existing types (a 'union')
- By setting the fixed attribute to true, you indicate that no derived types can later be defined with this type as its base.

Restrictions

- A simple way to form a new data type is to create a restricted version of an existing type.
- A **restriction** element is used to first identify, in its **base** attribute, the existing data type that is to be the basis of the new, more restricted type, then hold **enumeration** elements to identify each possible value.

```
<xs:simpleType name="securityLevel">
   <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="normal" />
      <xs:enumeration value="secret" />
      <xs:enumeration value="topSecret" />
   </xs:restriction>
</xs:simpleType>
```

Lists

- The **list** element is used to create lists of a given type. The *itemType* attribute identifies the data type that the values in the list must conform to.

```
<simpleType name="scoresList">
        <list itemType="integer">
</simpleType>
```

- Spaces separate each item in a list. The following example shows multiple integer values in a scores element that has adopted the type defined above
  - <scores>42 57 123 19</scores>
- It would be **dangerous** to base a list type on the 'string' data type, because strings can contain spaces. A string with one space in it would be treated as two list items.

Unions

- It is possible to create a new data type that is an amalgamation of two or more existing types. Values are considered valid when they conform to the constraints of any of the types concerned.
- The **union** element refers to two or more data types, using the **memberTypes** attribute to refer to the types to be included.

Unions example

```
<xs:simpleType name="noScore">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="none" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ScoreOrNoScore">
  <xs:union memberTypes="xs:integer noScore" />
</xs:simpleType>
<!--  validates <score>42</score>
        and <score>none</score>
-->
```

Hands on exercise 14

- Now do hands on exercise 14