XML-Schema Quick Start * Draft 2011-01

XML-Schema (quick start)

XML-Schema is a World Wide Web Consortium (W3C) specification for an XML *application* that is used to validate/constrain categories of XML documents. The XML-Schema has been adopted by industry and academia, world-wide as well as being a core component of many other XML specifications such as XPath, XQuery, XSLT, XPointer, XForms, SOAP, WSDL,UDDI, BPEL, Web Services, and SOA. An XML-Schema is a separate document or set of documents constructed by you to describe the allowable vocabulary and structure of the elements, attributes, and notations that make up an XML document. In fact, the conforming XML document is called an *instance* of the schema.(think of the 'bare' relational table schema versus the filled out tables with actual data that must conform to that schema, of if you know a programming language like Scala, Java, or C#, the class source file is the schema and the constructed object, with its accompanying specific data, is the instance.).

Once you have a schema, then you can invoke a *validating parser* that reads your *schema document* together with the XML document(s) to be checked, flagging any deviations. The parser equates and checks components of the XML document against components of the XML-Schema, based on shared *namespaces*, and in particular, *targetNamespaces*. The use of *namespaces* distinguishes multiple names in a document coming from different domains/ontologies and so having different meanings. (It is possible to not use namespaces at all, in very restricted circumstances, but these are very rare occasions).

In summary: XML-Schema provides the capability to validate data (structure, content, order, types, uniqueness, identity), serve as a contract between communicating agents, show stakeholders what the structure and data-types of a document must look like, provide application information during processing, provide system documentation, allow user extensions and derivations of new datatypes, and add in data for default and fixed values for both elements and attributes. (Whew!)

An Example Document and its Schema

Example 1.1 shows a complete XML document describing an individual *product* in terms of its' identifier, *id,* and its' *weight*. I have additionally shown linkage code directly in the document to allow a *validating parser* to identify the *schema* this document is to be validated against, namely, *product.xsd*. The use of namespaces in documents and schemas is the rule rather than the exception but initially, I will not use the full capabilities of this option. You will note the unavoidable use though, of distinguishing the vocabulary of the xml-schema tags, by using a prefix of *xsi*. By the way, every validating parser recognizes the XML-Schema namespaces and their associated vocabularies so you don't have to do any 'importing' of any additional files.

A ShortHand Notation for XML structures - use an indented outline to represent nesting

Before I code up an actual XML document with all the required angle bracket tags, and meta-data,

let me suggest a starter notation that can then be dressed up in all the necessary syntax. In short, use an indented list (shows the *structure*) together with an agreement as to how to show attributes and element *content*. Example 1.1, following this discussion, shows a final XML document with the required syntax, but, prior to that, you could get an idea of what you might want to do by using a simple outline as below. (I am using double slashes as in Java, JavaScript for comments).

```
product version=”2010-10-01” // use name=”value” pairs for attributes
      id         AB // follow the element tag name with the content
      weight     10.26 //follow the element tag name with the content
```

For each nested element, just indent, just like you do an outline for a school paper. In fact, for a ‘first look’, you might not even want to put in any example data, just the element names and attributes. To distinguish attribute values from element content, write the attributes as name value pairs, keeping the quotation marks. That is why I represented the attribute version as *version*=”2010-10-01”. If you don’t want to give the attribute a value yet, just use the empty string, like *version*=””

Example 1.1 an XML Document and Schema:

The XML document instance: product.xml

```
[1] <product version=”2010-09-01”
[2] xmlns:xsi = “http://www.w3.org/2001/XMLSchema-instance
[3] xsi:noNamespaceSchemaLocation = ‘product.xsd’>
[4] <id>ABC-123</id>
[5] <weight>10.26</weight>
[6] </product>
```

Line by line discussion of product.xml

[1] the product element, with a tag of *product*, has an attribute named *version* whose value is “2010-09-01”. This *date* format will be enforced in the *product.xsd* schema.

[2] *xmlns* is a built-in XML attribute that allows specification of a prefix, (*xsi* in this case) that can stand-in for an actual namespace which is “*http://www.w3.org/2001/XMLSchema-instance*”. That namespace is a *globally unique* string of characters (reserved by the World Wide Web Consortium (w3c)). The prefix is necessary since the actual namespace string contains forbidden characters such as a colon ‘:’ that would not be allowed in the XML document.

[3] *noNamespaceLocation* is an attribute in the *xsi* namespace whose value allows specification of a location to look for a validating schema. In this case, that validating schema is found in the same directory and is named *product.xsd.*

[4] *id* is an element with a tag of *id* and content of ABC-123. (this content is specified to be a *string* that could be further constrained in the schema, via a *regular expression*).

[5] weight is an element with a tag of *weight* and content of 10.26 (a number of type *double*). This value is further constrained in the schema to lie between 1.3 and 50.5 inclusive.

[6] the ending tag for the element *product*.

The XML-Schema Document: product.xsd

```
[1] <?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    elementFormDefault="qualified">
 <xsd:element name="product" type="productType"/>
[5]<xsd:complexType name="productType">
   <xsd:sequence >
     <xsd:element name="id" type="xsd:string"/>
     <xsd:element name="weight" type="weightType" />
   </xsd:sequence>
[10] <xsd:attribute name="version" type="xsd:string" />
 </xsd:complexType>
<xsd:simpleType name="weightType">
  <xsd:restriction base="xsd:double" >
    <xsd:minInclusive value="1.3" />
[15] <xsd:maxInclusive value="50.5" />
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Line by line discussion of product.xsd

[1] the standard XML declaration that says this is an XML file type, that conforms to the 1998 specification and will use character encoding in the format of UTF-8 (this is an efficient form of Unicode encoding). That is, the parser will use just enough bytes to encode the character, one byte for ASCII and up to 4 bytes for some asian languages)

[2] declare the element schema and its tag *schema* with its associated namespace. This namespace, "*http://www.w3.org/2001/XML-Schema*", delimits a set of tags of which: *schema*, *element*, *attribute*, . . are members. (Note that this set of terms/tags comprises a mini-ontology concerned with document validation.

[3] this constraint affects how the instance document must be tagged. This is relevant when using namespaces and dictates which tags must be namespace qualified and which not.

[4] declare an element of type *product* and declare that its type will be defined later, as a complexType called *productType*.

[5 -9] begin the definition of *productType* as a sequence of *id* and *weight* elements. The id element has a simple type of *xsd:string*. The weight element has a simple type, to be defined later, of *weightType*.

[10-11] line 10 defines the version attribute to have a date type which is built-in as *xsd:date*. Line 11 ends the definition of the *productType*.

[12] begin the definition of a simpleType, the weightType. Since weight is a number of type double, we start off by specifying that as the base, and restrict from there. That is, we are going to restrict a double number to lie between 1.3 and 50.5, inclusive.

Get Started Writing XML-Schemas

The next section is just a quick start introduction to XML-Schema. The intent is to get you up and running as fast as possible by using a few examples and a few reference tables. Later sections will go into detail on the topics introduced here. In a later section, (*see OOMetaphors**) I have drawn parallels to an object language like Java. For now though, take a look at the material below.

Types of Elements in XML-Schema and Related Instance Fragments

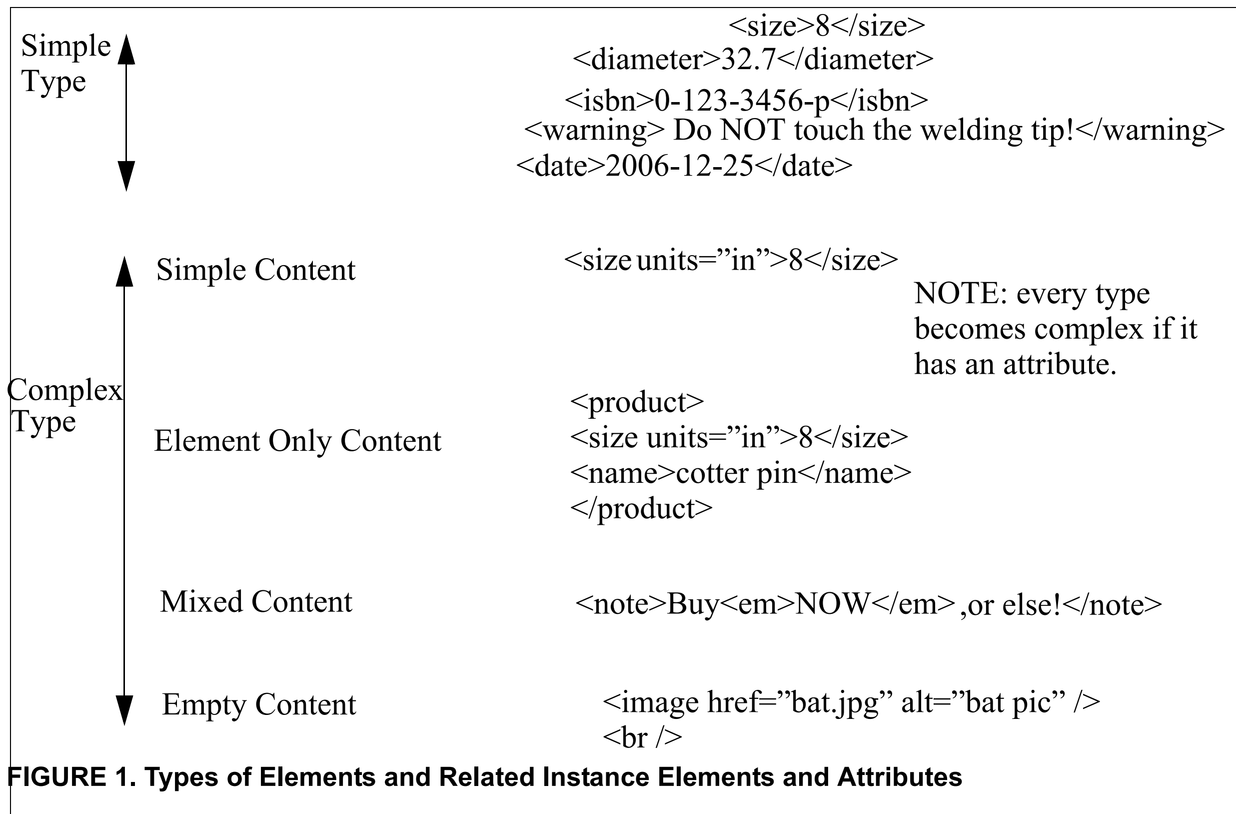

FIGURE 1. Types of Elements and Related Instance Elements and Attributes

Simple Types

Elements that have only character data as content, that is, just text strings with no attributes or any element children, are called *simple types*. This character data can have a type though, and it will be either a *primitive* type or one that you have derived by *restricting* one of the primitive types. It is also possible to have a simple type made up of a *list* or a *union* of simple types, although I won’t cover that here. Attributes cannot appear in simple element types. If there are attributes, then the element’s classification moves to a *complex* type.

Complex Types

Elements that have attributes are automatically complex. In addition, elements that have *element children* only, elements that have no content, that is, are *empty*, or elements that have both character and element children, that is, *mixed*, are all *complex*. A last category of complex types occurs when the content of the element is just character data, but it also has one or more attributes. This is called a complex type with *simple content*. (Note: if it were not for attributes, this last category would be a *simple type*). Below is a graphic of these four kinds of complex types. Complex types are further described at “Complex Types” on page 8.

Ex. S-2 Sample Complex Types

<size units="ft2" conditions="STP">27</size>	complex (with simple content)
<review>This is a really really neat book.</review>	complex (mixed)
<garden> <crop>corn</corn> <crop>peas</peas> </garden>	complex (element only children only)
<image href="myImage.jpg" alt="picture file" />	complex (empty content but two attributes)

FIGURE 2. The four kinds of complex types

Built-In Data Types

XML schema processors are required to handle the following datatypes by being able to detect if an XML document's element or attribute content conforms to it. If the associated schema declares that an element's content must be an *integer*, then the processor must report an error if the content is not an *integer*. There are 44 built in types of which maybe 20 are commonly used.

Category of Data Type	Built-in Data Types (I have left off some of the lesser used ones)
names and strings	string - general array of characters and white spaces token - special treatment of space collapse and white space stripping Name NCName - non colonized name (no colons ":" allowed) QName - qualified name language
numeric	float double decimal integer long, int, short, byte, positiveInteger, nonPositiveInteger, negativeInteger, nonNegativeInteger
dates and times	duration, dateTime, date, time, gYear, gYearMonth, gMonth, gMonthDay, gDay
utility	boolean, anyURI

Facets

These are constraints that are used with the simple types to do *restrictions* on a built in type or a simple type that I may have defined. Below is a table of the facets available within XML-Schema.

Category of Facet	Facet
limits	minInclusive maxInclusive minExclusive maxExclusive
length	length minLength maxLength
precision	totalDigits fractionDigits
enumerated valuesq	enumeration
patterns	pattern
whitespace processing	whiteSpace
Category of Facet	**Facet**

Regular Expressions

Regular Expressions make up a pattern language that is designed to select text strings from general text. Each pattern divides the searched text into two groups, those text strings that match and those that don't.

Metacharacter	Name	What it Matches
.	dot (period)	matches any character (usually doesn't match a new line however)
[ ]	character class	match any single character within these brackets
[^]	negated character class	match any character not listed inside the brackets
^	caret (circumflex)	match the position at the start of the line
$	dollar	match the position at the end of the line
\<	back slash less-than	match the position at the start of a word
\>	back slash greater-than	match the position at the end of a word
|	or bar	matches either expression it separates. This is the alternation symbol
( )	parentheses	groups expressions, usually to limit the scope of a metacharacter, or capture text for subsequent use. (See \1, \2... below)
\char	escaped character	matches the literal char Used to distinguish the literal role of a metacharacter.
? (this is appended to an item)	question mark	allow one or zero of the preceding item

Metacharacter	Name	What it Matches
* (this is appended to an item)	star (splat)	allow zero or more of the preceding item
+	plus	require 1 or more of the proceeding item
{min, max} {specific number}	specified range	require at least *min* and at most *max* of the preceding item.
\1, \2	back reference	matches text previously matched within the first, second, third, etc.. set of parentheses.
-	dash	multiple uses, depending on where it is used in the regular expression.
Metacharacter	**Name**	**What it Matches**

Making Your Own Simple Types (Restricting Primitives)

By definition, simple types can't be extended to include attributes, or contain additional elements as children. So, the only thing you can do is to restrict one of the built in types, or one of your own simple types.

Ex S-3 simple type using facets

```
<xs:simpleType name="sizeType">
        <xs:restriction base="xs:decimal">
            <xs:minInclusive value="1.1" />
            <xs:maxExclusive value="10.0"/>
        </xs:restriction>
  </xs:simpleType>
```

Ex S-4 simple type, restricting a user simple type

Now I take the sizeType defined above and further restrict it to one of two numbers, "2" or "4". This uses facets to further restrict a type I had defined.

```
<xs:simpleType name="sizeRestrictedType">
        <xs:restriction base="sizeType">
            <xs:enumeration value="2" />
            <xs:enumeration value="4" />
        </xs:restriction>
  </xs:simpleType>
```

Ex S-5 simple type, using the regular expression (regex) sublanguage

This example uses the pattern facet (a regular expression) to further restrict a general string datatype.

```
<xs:simpleType name="dressPattern">
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-D]{3}-[1-9]{2}-[SML]" />
        </xs:restriction>
  </xs:simpleType>
```

This regular expression (that is part of the simple type "dressPattern", would match the following content in an XML document:

AAA-12-L

ABC-99-S

It would not match the following since E and T are not in the allowable character sets:

BCE-89-T

Complex Types

Complex types allow you to define the content model and the attribute(s) of elements. This is another way of saying that an element has child elements or attributes or both. (There is a special case of an element having no attributes, children, or content. Such an element might simply be a 'mark-

er' in the text stream, perhaps indicating a error).

There are four categories of content that a ComplexType can hold: *simple content, element-only, mixed, and empty*. Examples were also shown previously in "The four kinds of complex types" on page 5.

Ex S- 5.2: Content Categories Within Complex Types

This next code snippet shows the four categories of content of a ComplexType.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
*Intent: show, in one file, the four categories of content for ComplexTypes
* simple, element-only, mixed, empty
*2010-11-30, r.r
-->
<moneyAlert>
   <currencyAmount currency="USD">23000</currencyAmount>
   <address country="US">
      <state>AZ</state>
      <city>Mesa</city>
   </address>
   <distribution>Attention: <title>Dr.</title> <name>Kim</name>,
   this report requires your <severity>immediate</severity>
   review. </distribution>
   <securityLevel value="orange" />
</moneyAlert>
```

Schemas for Complex Types

Following the discussion earlier, here are more examples of schemas for complex types. These schema fragments are designed to validate the elements and atributes within XML docs. For example, the first example, would validate an XML doc with an element tag named "garden" that was a container for "crop" elements, up to 100 of them in fact. Notice that the type of the crop element is declared as cropType but would be defined later in the document (or in another schema document).

Ex S-6 complex type, element only children

```
<xs:element name="garden" >
        <xs:complexType  >
                <xs:sequence minOccurs="1" maxOccurs="100">
                        <xs:element name="crop" type="cropType" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Ex S-7 complex type, empty with only attributes (although it doesn't even need attributes)

```
<xs:complexType name="imageType" >
                <xs:attribute name="href"  type="xs:anyURI" />
                <xs:attribute name="alt" type="xs:token" />
```

```
    </xs:complexType>
```

Ex S-8 complex type, mixed character data together with element children

The corresponding XML entry might have character content mixed in with tags such as:

. . . <review> What good is a reviewanyway? </review>.. . . <!-- an XML doc fragment->

```
<xs:complexType name="reviewType" mixed="true">
        <xs:sequence minOccurs="0" maxOccurs="100" >
        <xs:element name="em" type="xs:token" />
        </xs:sequence>
    </xs:complexType>
```

Ex S-9 complex type, simple content (character data content with attributes only)

Note: this complex type extends a simple type, from Ex S-3 by adding an attribute, “units”. This one is a little more interesting since it extends a type I created, that in turn was a *restriction* of a previous type, *sizeType*.

```
<xs:complexType name="sizeXXType">
        <xs:simpleContent>
            <xs:extension base="sizeRestrictedType">
            <xs:attribute name="units" type="xs:token" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>Identifiers as well
```

Identity and Key Constraints

These options for XML-Schema lets you uniquely identify nodes in your document and manage the integrity of the cross-references between them. Think of a relational table's keys. First off, the keys, which are attributes, must be *unique*. Secondly, the keys must be non-null. In XML-Schema this simply means that there *must be* a value. Finally, if you place a foreign key in your table, denoted usually as FK, then that foreign key must match a *key* elsewhere. NOTE: You will need to use an application program like XQuery to check this since XML-Schema won't.

The general structure is:

```
<xsd:key name = "myKeyForxxx>
      <xsd:selector xpath = an xpath that picks out elements to be
unique>
      <xsd:field xpath =an xpath that picks out the specific fields or
attributes within the above elements whose unique values effect the
uniqueness condition>
. . . more than one field/attribute can be defined that serves to
uniquely dientify elements
</xsd:key>
```

In our case, this would be defined as part of the element definition of the top most container, such as products, or movies, or employees, . . .

Identity Constraints

Uniqueness. this is a constraint that guarantees that a value or a composite set of values is unique within a *specified* scope. Scope will be illustrated below, but for now, just think of scope as being the enclosing container. Within that container, the values designated as unique, must be unique.

Key Constraints. this is a feature that enforces uniqueness and also requires that all values must be present. This is analogous to the requirement within the relational model where is something is a "key" it cannot be null or if it is a composite key, none of the components may be null.

Key References. this requires that a value or a composite of values corresponds to a value represented by a key. Again, this is like the Relational requirement that if a foreign key refers to a primary key, then that primary key must exist.

The general structure here is:

```
<xsd:keyref name= "myforeignKeyNameForxxx">
      <xsd:selector xpath = the xpath description that picks out a set
of elements>
      <xsd: field xpath = an xpath that picks out the specific fields or
attributes within the above elements whose unique values effect the
uniqueness condition>
. . . more than one field/attribute can be defined that serves to
uniquely dientify elements
</xsd:keyref>
```

Movies Example of Key and Foreign Key (*** NOTE: Validation only for single file)

Consider Movies XML file with the outlined structure of a standard overall container, *movies*, con-

taining multiple *movie* elements. Also I will show a related movie starts *stars XML* document and schema. The plan is to place a key on the *movies* file and a foreign key on the *stars* file that references the *movies key*. The *semantics* is that a movie star must have a reference to a movie they starred in. Think Employee must have a reference to a Department they 'starred' in.

> *****NOTE**: The scope of XML-Schema vaildation is a single doucment therefore, XML-Schema won't check related but separate documents.

XML Document Instance for Movies

```
movies
    movie
        title     (string)
        year      (integer)
        genre     (enumeration: comedy, drama, sciFi, western)
    movie
        title     (string)
        year      (integer)
        genre     (enumeration)
. . . //more movie here
```

XML Schema for Movies

XML-Schema for *movies* is straightforward. I am using an abbreviated notation here to avoid all the boilerplate angle brackest and prefixes.

```
element name = "movies" type = "moviesType"/>
complexType name = moviesType
        sequence
                movie maxOccurs ="300" // movie needs to be defined as well

        key name = "movieKey" // this key is defined within the moviesType
                selector xpath="movie"
                field xpath = "title"
                field xpath = "year"
*** end moviesType***
```

XML Document Instance for Movie Stars

```
stars
    star
        name
        address
        starredIn
            title
            year
```

star
    name
    address
    starredIn
        title
        year

. . . more star entries

XML-Schema for Movie Stars

element name=”stars” type = “starsType”
    complexType name = “starsType”
        sequence max Occurs = “300”
            star

Derivations of Complex Types

Once you have defined complex types, using the procedures discussed above, you can then further restrict or extend them. We can derive complex types from both simple types and complex types.

Why derive more types?

Subsetting. If you want to define a more *restrictive* subset of a schema then the way to go is *restriction* in a derivation. The result of a restriction derivation is a new type whose values are a subset of its parent. So, anywhere the parent works, the child works too.

Supersetting. If you want to *extend* your schema components, then the *extension* mechanism will work. This capability lets you add children, attributes, or both to a type. Note though, the base type values may not work as an extended type value since required elements and/or attributes may have been added.

Type Substitution. This is a way that derived types can substitute for their ancestor types in the XML documents.

Reuse. Of course, if you can reuse core functionality then do it. Knowing what is the core functionality is a question of design and ontology/domain knowledge but if you have that knowledge and option, then deriving from a common base is the way to go. (In some cases, derivation is not an option due to the schema being inaccessible or possibly organizational policy).

Deriving a complex type from a simple type or from a simple complex type

A `simpleContent` sub-element is used when you derive a complex type from a simple type or when you derive from a complex type that already has simple content.

Ex S 10.1 Deriving a complex Type from a simple type

Suppose I have already defined this frequencyType as a simple type, as below, but would now like to add in an attribute. That requires a complex type definition with an indication that the derivation is based on a simple type, *frequencyType*. The content is called simple because there is only character data plus an attribute. The addition of that attribute though, pushed the definition into the complex type domain. I call this new, derived type, *frequencyTypeX*.

```
<xs: simpleType name = “frequencyType” type = “xsd:integer” />

<xs:complexType name=”frequencyTypeX”>
<xs:simpleContent>
<xs:extension base=”frequencyType” >
<xs:attribute name = “units” type = “xs:token” use=”optional” />
</xs:extension>
</xs:simpleContent>
</xs:complexType>
```

Ex S 10.2 Deriving a complex type from a complex type having simple content

From example *S-10.1* I have derived a complex type from a simple type by adding in a attribute,

“units”.The result is called a complex type with *simple content.* So, I already have a complex type with simple content from example *S 10.1*. Now I will extend it further by adding in another attribute. *This illustrates a derivation of a complext type from a complex type with simple content.*

```
<xs:complexType name = “frequencyTypeXX >
<xs:simpleContent>
<xs:extension base= “frequencyTypeX”>
<xs:attribute name = “status” type=”xs:token” use=”required”/>
</xs:extension>
<xs:simpleContent>
</complexType>
```

Complex Content

A `complexContent` element is used to derive a complex type from another complex type that already has complex content. (Remember, these three kinds of content are characterized by element-only, mixed, and empty content types).

A complex derivation can add or remove parts of the content model as well as add or remove attributes.

Ex 10. 3 deriving a complex type from a complex type with element type content.

These kinds of extensions consist of adding to the content model or adding attributes

```
<xs:complexType name="ReportType">
   <xs:sequence maxOccurs="10">
      <xs:element name="distribution" type="xs:string" />
   </xs:sequence>
</xs:complexType>

<xs:complexType name="ReportTypeUK" >
   <xs:complexContent >
      <xs:extension base="ReportType">
      <xs:sequence maxOccurs="20">
      <xs:element name="dictionary" type="DictionaryType" />
      </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Checking a Document Using the Uniqueness Constraint

The document in Ex 11.1, a Glossary, must have unique entries as far as it's terms. If there are multiple authors working on a document, which is not that unusual a circumstance, the current author won't always be able to know if a word as been defined earlier, or elsewhere. Further, using the *include* capabilities of XML, the current author will know even less about the content of entries coming in over the net. So, under these circumstances, it would be helpful for the system to check the assembled document for duplicated terms. The document below shows such a situation and it will be followed by a schema, Ex 11.2, that will flag this error.

Ex 11.1: Glossary with Duplicate Terms

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- glossaryduplicates.xml
*Intent: to show how duplicates can be caught by
* a schema. See the validating schema glossaryduplicates.xsd
*2003-10-27
-->
<glossary>
   <glossentry>
   <glossterm>CSS              </glossterm>
   <glossdefinition>Cascading Style Sheets is a lightweight, client side,
    web targeted styling language used
   as an adjunct to HTML and XML documents.
   </glossdefinition>
   </glossentry>
 <glossentry>
   <glossterm>XForms</glossterm>
   <glossdefinition>A recent W3C recommendation (Oct. 14, 2003)
    that is an XML-based rewrite of HTML forms.
   </glossdefinition>
   </glossentry>
   <glossentry>
   <glossterm>XForms</glossterm>
   <glossdefinition>A recent W3C recommendation (Oct. 14, 2003)
    that is an XML-based rewrite  of HTML forms.
   </glossdefinition>
   </glossentry>
</glossary>
```

Ex 10.2: The Validating Schema for GlossaryDuplicates

The schema below, when executed, will flag the duplicate `glossterm` found in Ex 11.1. The syntax of the constraint is described in three points as follows::

scope. . the scope is the extent of the effect of the constraint. To determine this extent, you place the constraint code in the highest level element that covers the effect you want. In the example document glossaryduplicates.xml, the scope I want is the whole document, so I place the constraint within the root element `glossary`, as shown in the code below.

selector. the selector selects all the nodes to which the constraint applies. This is an XPATH expression that takes as its *context* node the root element, `glossary`. In the code below, the selector "glossentry" will construct a node set consisting of the three nodes in the XML document labeled

`glossentry`. It is these three nodes that I want to be unique. To do that, I use the *field* value of one of the children, `glossterm`, as a differentiator.

field. the last piece of the uniqueness constraint is the actual values to be checked in elements and attributes. In the example document, this *field* is the value of the `glossterm` element. In this case it is the text string “XForms” that is duplicated between nodes. Note that I can specify multiple fields and attributes and then use their combinations to check for uniqueness.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
   <xs:element name="glossary" type="GlossaryType" >
   <xs:unique name="glossKey">
      <xs:selector xpath="glossentry" />
      <xs:field xpath="glossterm" />
   </xs:unique>
   </xs:element>

   <xs:complexType name="GlossaryType">
         <xs:sequence>
            <xs:element name="glossentry" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="glossterm" type="xs:string"/>
                     <xs:element name="glossdefinition" type="xs:string"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
```

Key Constraints

These constraints are very close to the uniqueness constraints discussed above. Again, the combination of the field(s) in the key must be *unique*, but even more, those fields must *exist*. Note that this differs from the uniqueness constraint illustrated in Ex 11.1 and Ex 11.2 above which *didn't* require existence of the comparing fields.

This means that you can't have any optional attributes or elements or nillable fields. These key constraints are effected by a key element whose syntax is identical with the unique element. Next is a fragment using a key constraint on the same document, glossary.xml.

Ex 11.3 A Key Constraint on the Glossary Document

Next is a fragment of a schema showing the (minor) syntactical changes are needed to change from a uniqueness constraint to a key constraint. Compare Ex 11.2 with this example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- keyconstraintfragment.xml
*Intent: show a fragment of a key constraint
* See also glossaryduplicates.xml / .xsd
*2003-10-28
-->
<!-- . . . additional schema code here           -->
```

```
<xs:element name="glossary" type="GlossaryType" >
   <xs:key name="glossaryentrykey">
      <xs:selector xpath="glossentry" />
         <xs:field xpath="glossterm" />
   </xs:key>
   <!-- . . .  additional schema code here   -->
</xs:element>
```

Key References

This capability lets you make sure that cross-references are valid *within* our instance document. These are like foreign keys in a relational database.

Element Declarations

The workhorses of an XML document are its elements and attributes. The table below, ****, shows the attributes that can appear within the element tag. See the following table ****, for the *content* that may appear between the start and end `element` tags. Element declarations assign element-type names and data types to elements.The element declarations can be either *global* or *local*.

Global Declarations

These declarations come immediately under the top level `schema` element. That is, their parent must be `schema`. The purpose of declaring elements globally is that they may then be called or referenced from other elements. Note that the names you choose must be unique throughout the schema, even if the schema contains other imported schema documents. The name you use for an element must obey the XML rules: it must be non-colonized, it must start with a letter or underscore and subsequent characters can only be: underscores, hyphens, numbers, letters and periods.

Element-type names are *automatically* qualified by the target namespace of the schema document and so its not legal to attach a prefix to the *`name`* attribute value. If you need one of your elements to apply to a different namespace then you will have to create another schema document and import that. This is in fact, what was done in "Ex 2.1: GlossarySales XML Document" on page 29. There, a `salesVolume` element from another namespace was specified and so a separate schema, defining `salesVolume`, was constructed and then imported to the "main" schema. (See "Ex 2.3: An Imported Schema For the salesVolume Element" on page 6 for this additional imported schema.)

Allowable Attributes in Global Element Declarations

Global elements are those that are direct children of the schema element. The attributes that are allowable within global elements are shown in the next table.

Attribute Name	**Type**	**Required/ Default**	**Discussion**
abstract	boolean		this element can't be used directly in an instance but can be the parent of a derived element.
block	#all |list of (substitution, extension, restriction)	blockDefault of schema	should type or element substitutions be blocked from the instance
default	string		processor inserts this value if the elements content is empty.
final	#all | list of extension | substitution	finaldefault of schema	allows the author to constrain the use of this attribute relative to: extension, restriction, head of a substitution group
fixed	string		processor checks inserted content for a match
id	ID		unique ID as in the DTD specification
name	NCName		non-colonized (no colons ":" allowed)
nillable	boolean	false	can xsi:nil be used in an instance?
type	QName		data type
Attribute Name	**Type**	**Required/ Default**	**Discussion**

An example here might be:

<xs:element name="product" id="p51" type="ProductType" > . . . content . . . <?product>

Global Element Content

Between the global element tags you can potentially see the following elements: What this says is that between the start and end tags of the element you can have:

- annotation -optional
- an optional simple type of complex type (or neither)
- zero or more `key`, `keyref`, or `unique` elements

Possible Content of the Global Element
annotation?, (simpleType | complexType)?, (key | keyref | unique)*

Local Element Declarations

These are the entities that are totally inside of a `complexType` definition. These declarations can only be used y the enclosing type and can't be referenced from elsewhere or participate in a substitution group. Example 7.1, shows local element declarations. Both `name` and `frequency` are totally within the complexType `SalesReport` and so can't be referenced by anything outside of SalesReport. Notice that local elements can have min. and max occurrences attributes.

Ex. S 7.1 A local element declaration inside of a global complex type.

In this case, the complex type "SalesReport" contains two locally declared elements, `name` and `frequency`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="SalesNS"
xmlns="SalesNS">
<xs:complexType name="SalesReport">
<xs:sequence >
   <xs:element name="name"  type="xs:string" />
   <xs:element name="frequency" type="xs:integer" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
</xs:schema>
```

Local Element Attributes

The next table shows what attributes are possible with local elements. Notice that there is a difference between these and the global element declarations.

Attribute Name	Type	Required/ Default	Discussion
block	#all |list of (substitution, extension, restriction)	blockDefault of schema	should type or element substitutions be blocked from the instance
default	string		processor inserts this value if the elements content is empty.
Attribute Name	**Type**	**Required/ Default**	**Discussion**

Attribute Name	Type	Required/ Default	Discussion
fixed	string		processor checks inserted content for a match
id	ID		unique ID as in the DTD specification
minOccurs	non-negative integer	1	min. number of times an element can appear
maxOccurs	nonNegative integer		
name	NCName	required	non-colonized
nillable	boolean	false	can xsi:nil be used in an instance?
type	QName		data type
Attribute Name	**Type**	**Required/ Default**	**Discussion**

Local Element Content

Between the local element tags you can potentially see the following elements (same as Global elements):

Possible Content of the Local Element
annotation?, (simpleType | complexType)?, (key | keyref | unique)*

In the case of the `product` element illustrated above, we have seen that it is an *element only* type and so its content model must consist of a `complexType`. Of course, within that child complexType there can be recursive children types either simple of complex type.

Attribute Declarations

Attributes within elements are the second most important component within XML documents. Attributes are declared similarly to elements. Lets declare an attribute that will be used as a component of a `loanAmount` element that will be shown next. We would like the attribute to be able to take a restricted set of currency designations such as USD for United States Dollar, EU for the European Union currency, CAN for the Canadian Dollar, and MEX for the Mexican Peso. This restricted set of simple values is called an *enumeration*.

Ex S 8.1: A SimpleType as an Enumeration

The file below, includes the element `loanAmout` that has an attribute, *`currencyType`* that is a simpleType consisting of an enumeration. We will allow the following values, that is, an enumeration, for the currencyType attribute: USD, CAN, MEX, CAN. Note that the instance document doesn't show all these, but they are all allowed. See the validating schema next for how to set up the enumeration.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- simpletypesenumeration.xml
*Intent: show how simple types work, together with their attributes
* this example shows enumeration
*2003-10-25
-->
```

```
<LoanDetail xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation= "simpletypesenumeration.xsd">
   <loanAmount currencyType="CAN">12345.45</loanAmount>
   <loanAmount currencyType="MEX">45000.21</loanAmount>
</LoanDetail>
```

Note that the `LoanDetail` calls out the location of the validating schema for this document, which was then successfully invoked. The "xsi:noNamespaceSchemaLocation" indicates that the validating schema has no target namespace.

Ex 8.2: The Validating Schema for "A SimpleType as an Enumeration"

The schema below validates the document contained in the file **simpletypesenumeration.xml**, which is Ex S 8.1. Note the coding style which is called the "Russian Doll" pattern.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--simpletypesenumeration.xsd
* Show an enumeration for an attribute-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDe-
fault="qualified">
   <xs:element name="LoanDetail">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="loanAmount" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:simpleContent>
                     <xs:extension base="xs:decimal">
                        <xs:attribute name="currencyType" use="required">
                           <xs:simpleType>
                              <xs:restriction base="xs:string">
                                 <xs:enumeration value="CAN"/>
                                 <xs:enumeration value="MEX"/>
                                 <xs:enumeration value="USD"/>
                                 <xs:enumeration value="EU"/>
                              </xs:restriction>
                           </xs:simpleType>
                        </xs:attribute>
                     </xs:extension>
                  </xs:simpleContent>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

A couple of comments on this schema might be helpful. First off, you can see it is an example of the direct specification of each element using anonymous types for all the children of the root element. This is understandable but pretty verbose. We could do better by declaring some of the types indirectly as in the next schema.

A Re-Written Schema for "A SimpleType as an Enumeration"

```
<?xml version="1.0" encoding="UTF-8"?>
<!--simpletypesenumerationrewritten.xsd
```

```
* This is a rewrite of the simpletypesenumeration.xsd
* to show how indirect type declarations can improve
* the appearance ( depending on your perspective!)
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDe-
fault="qualified">
   <xs:element name="LoanDetail" type="LoanDetailType"/>

   <xs:complexType name="LoanDetailType">
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
         <xs:element name="loanAmount" type="loanAmountType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="loanAmountType">
      <xs:simpleContent>
         <xs:extension base="xs:decimal">
               <xs:attribute name="currencyType" type="currencyTypeType"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>

   <xs:simpleType name="currencyTypeType">
      <xs:restriction base="xs:string">
         <xs:enumeration value="CAN"/>
            <xs:enumeration value="MEX"/>
         <xs:enumeration value="USD"/>
         <xs:enumeration value="EU"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

A Table of `Attribute` Attributes for Use Within a Schema

Next is shown the attributes that the `attribute` element can have.

Attribute	Description and Discussion
default	this is the attribute and its value that is inserted by the processor if the element type does not have this attribute explicitly specified.
fixed	when the element in an instance does specify this attribute, it must have this value.
id	this value uniquely identifies an entity within the whole schema
name	this is the name of the attribute that can be used to reference this attribute within the rest of the schema
type	this is the simple type of the attribute. Note: attributes are only of simple type.
use	this value determine if the attribute is required, prohibited, or optional
ref	this value refers to a global attribute type. This attribute, like the corresponding ref attribute for elements, allows multiple elements to specify the same global attribute type.
Attribute	**Description and Discussion**

Simple Types

Simple types come from the built in primitives, derivations from those, or user derivation from

simple types. A simple type *restricts* the value of an element's content or an attribute's value. Simple types mean that the data under consideration can't be split up further within the XML-Schema types. For example, a sequence of characters is a "string" in XML-Schema and can't be reduced further within XML-Schema, even though it is composed of characters. Below is an example of a simpleType. `frequency` is an element with a simpleType of content.

```
<frequency>60</frequency>
```

Attributes can only have simpleType content. The next example shows an element with attributes: *`red`*, *`green`*, *`blue`*, and *`date`*. All have simpleTypes as their values, with the first three being integers and the last attribute having the *`date`* simpleType.

<color red="255" green="0" blue="0" date="2003-10-25"/>

Additional restrictions are made by using constraining *facets*. (See "Simple Type, Restriction Sub-elements (facets)" on page 19, for a list of potential constraining facets that can be used with the `restriction` element.)

A Schema Example Showing A Simple Type with Facets

The XML-Schema document shown next illustrates the process restricting an integer base type to a range from 1 to 30,000 inclusive. This is a case of a simple type being derived from another simple type.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- simpletypes.xsd
*Intent: show how simple types work, together with their attributes
*2003-10-20
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="restrictedFrequency" type="FrequencyType" />

<xs:simpleType name="FrequencyType">
   <xs:restriction base="xs:integer">
      <xs:maxInclusive value="30000" />
      <xs:minInclusive value="1" />
   </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Element	Description and Discussion
annotation	this is a documentation element
enumeration	specify a constant value. This is like the enumeration type within programming languages. For example, the t-shirt sizes can be small, medium, large. Those three values would comprise an enumeration.
fractiondigits	determines the number of digits after the decimal in a decimal number
length	specify the number of characters in a string, after normalization (white space elimination)
maxExclusive	specify an numeric upper bound, including the bounding number.
maxInclusive	specify an numeric upper bound, not including the bounding number.
minExclusive	specify an numeric lower bound, not including the bounding number.
minInclusive	specify an numeric lower bound, including the bounding number.
Element	**Description and Discussion**

Element	Description and Discussion
minLength	specify the minimum number of characters in a string, after normalization (white space elimination)
pattern	this value contains a *regular expression*
totalDigits	this is the total number of digits allowed in a decimal number.
simpleType	this element specifies a local simple type
whitespace	specifies how white space (space, carriage return, line feeds and tabs) are to be treated.
Element	**Description and Discussion**