**XML**

XML is a text-based mark-up language that is the de facto industry standard for exchanging data among disparate applications. XML defines precise syntactic rules for what constitutes a well-formed XML document.

Syntactic rules defining an XML document:

* First, all syntactic constructs within an XML document are delimited by mark-up character sequences, which imply that within the body of any syntactic construct, the mark-up character sequences are not allowed. For example, a syntactic construct called a start tag is delimited by < and > characters, which implies that these two characters cannot appear within the body of a start tag.
* Second, if you need to get around the limitation described in the previous bulleted item, escape character sequences allow you to do that.

**XML Declarations:**

A well-formed XML document can begin with an XML declaration. An XML declaration can be omitted, but if it appears, it should be the first thing within a document. You define an XML declaration as follows:

<? xml version='1.0' ?>

The version attribute specifies the XML version, and it is a required attribute. The XML declaration may include additional attributes: encoding and standalone. An example XML declaration with the encoding and standalone attributes is as follows:

<? xml version='1.0' encoding='UTF-8' standalone='yes' ?>

The encoding attribute specifies the character set used to encode data in an XML document. The default encoding is UTF-8. The standalone attribute specifies whether the XML document references external entities. If no external entities are referenced, specify the standalone attribute as yes.

**Elements:**

The basic syntactic construct of an XML document is an element. An element in an XML document is delimited by a start tag and an end tag. An example of an XML element is as follows:

<journal></journal>

A start tag within an element is delimited by the < and > characters and has a tag name. In the previous start tag, the name is journal. The precise rules for a valid tag name are fairly complex and best left to the W3C Recommendation. However, it is useful to keep in mind that a tag name must begin with a letter and can contain hyphen (-) and underscore (\_) characters. An end tag is delimited by the </ and > character sequences and also contains a tag name.

A document must have a single root element, which is also known as the document element. If you assume that the journal element is your root element, then your document so far looks as follows:

<? xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<journal></journal>

Element text content cannot contain any delimiter character sequences such as </.One way to get around that is to enclose element content within a CDATA construct, and assuming you do that for this example, your document now looks as follows:

<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<journal>

<article>

<![CDATA[This is some arbitrary text <within> a CDATA!]]>

</article>

</journal>

Now let’s assume you want to add another attribute named date with the value <04/12/2006>.If you recall the first central concept we mentioned at the outset of this primer, you are not allowed to include delimiter characters within an attribute value. However, the second central concept mentioned earlier comes to your rescue: you can use the &lt; character sequence to escape <, and—yes, you guessed it—you can use the &gt; character sequence to escape >. So, with that in place, thedocument now looks as follows:

<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<journal>

<article date="&lt;04/12/2006&gt;" title="A Tutorial on XML 1.0" >

<![CDATA[This is some arbitrary text <within> a CDATA!]]>

</article>

<article/>

</journal>

Another mechanism for including delimiter characters within the body of a construct is to use escaped numeric references. For example, the numeric American Standard Code for Information Interchange (ASCII) value for the > character is 62, so you can use the &#62; character sequence instead of &gt;. Using escaped numeric references is of course the most general mechanism for including delimiter characters within a construct’s body.

**Comments:**

You can define comments in an XML document within a comment declaration as shown in the following example:

<!--This is a comment - ->

Comments can appear anywhere outside markup, which consists of start tags, end tags, empty element tags, comments, CDATA sections, escape character references, and entity references.

**DOCTYPE Declarations:**

An XML document can also include a document type definition (DTD).A DTD defines the structure of an XML document. If the content of an XML document conforms to the structure imposed by its DTD, then such a document is termed valid. A DTD is defined in a DOCTYPE declaration. A DOCTYPE has three types of DTD specifications: internal, private, and public. You can specify an internal DTD within an XML document as follows:

For example, you could have an internal DTD for the example document as shown here:

<!DOCTYPE journal

[

<!ELEMENT journal (article)\*>

<!ELEMENT article (#PCDATA)>

<!ATTLIST article title CDATA #IMPLIED>

]>

You can specify a private external DTD as follows:

<!DOCTYPE rootElement SYSTEM "DTDLocation">

For example, assuming a DTD for the example document exists in a local file named journal.dtd, you can specify a private external DTD as shown here:

<!DOCTYPE journal SYSTEM "journal.dtd">

You can specify a public external DTD as follows:

<!DOCTYPE rootElement PUBLIC "DTDName" "DTDLocation">

You can specify a public external DTD as shown here:

<!DOCTYPE journal PUBLIC "-//Apress.//DTD Journal Example 1.0//EN"

"http://www.apress.com/javaxml/dtd/journal.dtd">

**Namespaces in XML:**

An XML Namespace associates an element or attribute name with a specified URI and thus allows for multiple elements (or attributes) within an XML document to have the same name yet have different semantics associated with those names because they belong to different XML Namespaces. The key point to understand is that the sole purpose of associating a uniform resource indicator (URI) to a namespace is to associate a unique value with a namespace. There is absolutely no requirement that the URI should point to anything meaningful. You specify an XML Namespace through one of two reserved attributes:

* You can specify a default XML Namespace URI using the xmlns attribute.
* You can specify a nondefault XML Namespace URI using the xmlns:prefix attribute, where prefix is a unique prefix associated with this XML Namespace.

**XML Schema:**

The XML Schema definition language specifies the structure of an XML document and constrains its content. The key concept to understand is that a schema based on the XML Schema language defines a class of valid XML documents. A document is considered valid with respect to a schema if it conforms to the structure defined by the schema. A valid XML document is formally referred to as an instance of the schema document. As a rough analogy, what a Java class is to a Java object, a schema is to an XML document.

One more important point to keep in mind is that a schema is also an XML document. In fact, this was one of the key motivations for the XML Schema language; the alternative structure standard, which is a DTD, is not an XML document. In case it is not already obvious, you could actually write a schema for an XML Schema–based schema document!

**Schema Declarations:**

The root element of a schema is schema, and it is defined in the XML Schema namespace xmlns:xsd="http://www.w3.org/2001/XMLSchema". An example schema document with its root element is as follows:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

</xsd:schema>

**Built-in Datatypes:**

The XML Schema language has 44 built-in simple types. These data types of course belong to the XML Schema namespace, so we will use them with the xsd: prefix, as in xsd:string. Commonly used Built-in data types are string, int , double, decimal, date, time.

**Element Declarations:**

You define an element in an XML Schema–based schema with the element construct, as shown here:

<xsd:element name="element\_name" type="element\_type"/>

You can define an element within a schema construct. The example schema document with atop-level catalog element declaration within a schema construct is as follows:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

<xsd:element name="catalog" type="catalogType" ></xsd:element>

<!-- we have yet to define a catalogType -->

</xsd:schema>

Of course, we have not yet defined catalogType. The XML Schema language defines two main type constructs: a simple type and a complex type. Almost no meaningful document structure is feasible without the use of a complex type.

**Complex Type Declarations:**

A complexType constrains elements and attributes in an XML document. You can specify a complexType in a schema construct or an element declaration. If you specify a complexType in a schema construct, the complexType is referenced in an element declaration with a type attribute. In the example schema, you can define the catalogType type as a complex type as shown here:

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

<xsd:element name="catalog" type="catalogType" ></xsd:element>

<xsd:complexType name="catalogType" >

</xsd:complexType>

</xsd:schema>

* **Sequence Model Groups**:

You can also define an element within a sequence model group, which, as the name implies, defines an ordered list of one or more elements. In the example schema, say you want to allow a journal element in the catalogType complex type; you’d use a sequence model group as shown here:

<xsd:complexType name="catalogType" >

<xsd:sequence>

<xsd:element ref="journal" />

<!-- we have yet to define a global journal element -->

</xsd:sequence>

</xsd:complexType>

The journal element declaration within the catalogType complex type uses a ref attribute to

refer to a global journal element definition. Of course, we have not yet defined any global journal element, so we will do that next, using a choice model group.

* **Choice Model Groups:**

You can also define an element within a choice model group, which defines a choice of elements from which one element may be selected. In the example schema document, say you want to define a global journal element that offers a choice between article and research elements, as shown here:

<xsd:element name="journal" >

<xsd:complexType>

<xsd:choice>

<xsd:element name="article" type="paperType" />

<xsd:element name="research" type="paperType" />

<!-- we have yet to define a paperType type -->

</xsd:choice>

</xsd:complexType>

</xsd:element>

* **All Model Groups:**

You can also define an element within an all model group, which defines an unordered list of

elements, all of which can appear in any order, but each element may be present at most once. In theexample schema document, you can define the paperType complex type with an all model group, as shown here:

<xsd:complexType name="paperType" >

<xsd:all>

<xsd:element name="title" type="titleType" />

<xsd:element name="author" type="authorType" />

<!-- we have yet to define titleType and authorType -->

</xsd:all>

</xsd:complexType>

* **Named Model Groups:**

You can define all the model groups you’ve seen so far—sequence, choice, and all—within a named model group. The named model group in turn can be referenced in complex types and in other named model groups. This promotes the reusability of model groups. For example, you could define paperGroup as a named model group and refer to it in the paperType complex type using the ref attribute, as shown in the following example:

<?xml version='1.0' encoding='UTF-8' ?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:complexType name="paperType">

<xsd:group ref="paperGroup" />

</xsd:complexType>

<xsd:group name="paperGroup">

<xsd:all>

<xsd:element ref="title" />

<xsd:element ref="author" />

</xsd:all>

</xsd:group>

* **Attribute Declarations:**

You can specify an attribute declaration in a schema with the attribute construct. You can specify an attribute declaration within a schema or a complexType. For example, if you want to define the title and publisher attributes in the catalogType complex type, you can do so as shown here:

<xsd:complexType name="catalogType">

<xsd:sequence>

<xsd:element ref="journal" minOccurs="0" maxOccurs="unbounded" />

</xsd:sequence>

<xsd:attribute name="title" type="xsd:string" use="required" />

<xsd:attribute name="publisher" type="xsd:string" use="optional" default="Unknown" />

</xsd:complexType>

An attribute declaration may specify a use attribute, with a value of optional or required. The default use value for an attribute is optional. In addition, an attribute can specify a default value using the default attribute, as shown in the previous example. When an XML document instance does not specify an optional attribute with a default value, an attribute with the default value is assumed during document validation with respect to its schema. Clearly, an attribute with a default value cannot be a required attribute.

* **Attribute Groups:**

An attributeGroup construct specifies a group of attributes. For example, if you want to define the attributes for a catalogType as an attribute group, you can define a catalogAttrGroup attribute group, as shown here:

<xsd:attributeGroup name="catalogAttrGroup" >

<xsd:attribute name="title" type="xsd:string" use="required" />

<xsd:attribute default="Unknown" name="publisher" type="xsd:string" use="optional" />

</xsd:attributeGroup>

You can specify an attributeGroup in a schema, complexType, and attributeGroup. You can

specify the catalogAttrGroup shown previously within the schema element and can reference it using the ref attribute in the catalogType complex type, as shown here:

<xsd:complexType name="catalogType" >

<xsd:sequence>

<xsd:element maxOccurs="unbounded" minOccurs="0" ref="journal" />

</xsd:sequence>

<xsd:attributeGroup ref="catalogAttrGroup" />

</xsd:complexType>

* **Simple Content:**

A simpleContent construct specifies a constraint on character data and attributes. You specify a simpleContent construct in a complexType construct. Two types of simple content constructs exist: an extension and a restriction.You specify simpleContent extension with an extension construct. If you want to define an authorType as an element that allows a string type in its content and also allows an email attribute, you can do so using a simpleContent extension that adds an email attribute to a string built-in type, as shown here:

<xsd:complexType name="authorType" >

<xsd:simpleContent>

<xsd:extension base="xsd:string" >

<xsd:attribute name="email" type="xsd:string" use="optional" />

</xsd:extension>

</xsd:simpleContent>

</xsd:complexType>

You specify a simpleContent restriction with a restriction element. If you want to define a

titleType as an element that allows a string type in its content but restricts the length of this content to between 10 to 256 characters, you can do so using a simpleContent restriction that adds the minLength and maxLength constraining facets to a string base type, as shown here:

<xsd:complexType name="titleType" >

<xsd:simpleContent>

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

<xsd:minLength value="10" />

<xsd:maxLength value="256" />

</xsd:restriction>

</xsd:simpleContent>

</xsd:complexType>

* **Complex Content:**

A complexContent element specifies a constraint on elements (including attributes). You specify a complexContent construct in a complexType element. Just like in the case of simple content, complex content has two types of constructs: an extension and a restriction.You specify a complexContent extension with an extension element. If, for example, you want to

add a webAddress attribute to a catalogType complex type using a complex content extension, you can do so as shown here:

<xsd:complexType name="catalogTypeExt" >

<xsd:complexContent>

<xsd:extension base="catalogType" >

<xsd:attribute name="webAddress" type="xsd:string" />

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

You specify a complexContent restriction with a restriction element. In a complex content restriction, you basically have to repeat, in the restriction element, the part of the base model you want to retain in the restricted complex type. If, for example, you want to restrict the paperType complex type to only a title element using a complex content restriction, you can do so as shown here:

<xsd:complexType name="paperTypeRes" >

<xsd:restriction base="paperType" >

<xsd:all>

<xsd:element name="title" type="titleType" />

</xsd:all>

</xsd:restriction>

</xsd:complexType>

A complex content restriction construct has a fairly limited use.

* **A complete schema example:**

<?xml version='1.0' encoding='UTF-8' ?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="catalog" type="catalogType" />

<xsd:complexType name="catalogType">

<xsd:sequence>

<xsd:element maxOccurs="unbounded" minOccurs="0" ref="journal" />

</xsd:sequence>

<xsd:attribute name="title" type="xsd:string" use="required"/>

<xsd:attribute default="Unknown" name="publisher" type="xsd:string" />

</xsd:complexType>

<xsd:element name="journal">

<xsd:complexType>

<xsd:choice>

<xsd:element name="article" type="paperType"/>

<xsd:element name="research" type="paperType"/>

</xsd:choice>

</xsd:complexType>

</xsd:element>

<xsd:complexType name="paperType">

<xsd:all>

<xsd:element name="title" type="titleType"/>

<xsd:element name="author" type="authorType"/>

</xsd:all>

</xsd:complexType>

<xsd:complexType name="authorType">

<xsd:simpleContent>

<xsd:extension base="xsd:string">

<xsd:attribute name="email" type="xsd:string" />

</xsd:extension>

</xsd:simpleContent>

</xsd:complexType>

<xsd:complexType name="titleType">

<xsd:simpleContent>

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

<xsd:minLength value="10"/>

<xsd:maxLength value="256"/>

</xsd:restriction>

</xsd:simpleContent>

</xsd:complexType>

</xsd:schema>