Validating XML

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

Should I take this tutorial?

This tutorial examines the validation of XML documents using either Document Type Definitions (DTDs) or XML Schema. It is aimed at developers who have a need to control the types and content of the data in their XML documents, and assumes that you are familiar with the basic concepts of XML. (You can get a basic grounding in XML itself through the *Introduction to XML* tutorial.) It also assumes a basic familiarity with XML Namespaces. (You can pick up the basics of namespaces in the *Understanding DOM* tutorial.)

This tutorial demonstrates validation using Java from the command line, but the principles and concepts of validation are the same for any programming environment, so Java experience is not required to gain a thorough understanding. DTDs and XML Schema, in particular, are language- and platform-independent.

What is XML validation?

In the creation of a database, using a data model in conjunction with integrity constraints can ensure that the structure and content of the data meet the requirements. But how do you enforce that kind of control using XML, when your data is just text in hand-editable files? Fortunately, validating files and documents can make sure that data fits constraints. In this tutorial, you'll learn what validation is, and you'll learn how to check a document against a Document Type Definition (DTD) or an XML Schema document.

DTDs were originally defined in the XML 1.0 Recommendation and are a carryover from the original Standard Generalized Markup Language (SGML), the precursor to HTML. Their syntax is slightly different from XML, which is one drawback to using them. They also have limitations in how they can be used, which led developers to seek an alternative in the form of XML schemas. DTDs are still in use in a significant number of environments, however, so an understanding of them is important.

There are a number of competing schema proposals, but the primary alternative to DTDs is the XML Schema Recommendation maintained by the W3C. (Throughout the course of the tutorial, "XML Schema" should be considered synonymous with "W3C XML Schema.") These schema documents, which, in terms of syntax, are also XML documents, provide a more familiar and more powerful environment in which to create the constraints on the data that can exist in XML data.

Please note that validation is by no means a requirement when working with XML data. If the overall structure and content of the XML data aren't crucial, feel free to bypass validation.

By the end of this tutorial you will learn how to create both a DTD and an XML Schema document. You'll also learn the concepts of using them to validate an XML document.

Tools

This tutorial will help you understand the topic even if you read the examples rather than trying them out. If you do want to try the examples as you go through this tutorial, make sure

you have the following tools installed and working correctly:

* A text editor: XML files, DTDs, and XML Schema documents are simply text. To create and read them, a text editor is all you need.
* JavaTM 2 SDK, Standard Edition version 1.3.1: XML can be manipulated and validated in any language where a validating parser is available. The bulk of the tutorial deals with the creation of documents, but you will also see how to build an application that uses a validating parser. To execute these examples, you also need a Java environment such as the JavaTM 2 SDK (available at *http://java.sun.com/j2se/1.3/* , and a validating parser).
* A validating parser: There are numerous validating parsers on the market. This tutorial demonstrates two Java versions: Xerces 1.4.2, which supports XML Schema, and JAXP 1.1, Sun's reference implementation. Xerces can be downloaded from *http://xml.apache.org/xerces-j/index.html* , and JAXP can be downloaded from *http://java.sun.com/xml/xml_jaxp.html* .

If you have a different set of tools installed, you can use them instead. Just check the documentation for instructions on turning on validation. You can download C++ and Perl implementations of Xerces from the Apache Project at *http://xml.apache.org* .

Conventions used in this tutorial

There are a few conventions that are used in this tutorial to reinforce the material at hand:

* Text that needs to be typed is displayed in a **`bold monospace`** font. In some code examples, bold is used to draw attention to a tag or element being referenced in the accompanying text.
* *Emphasis/Italics* is used to draw attention to windows, dialog boxes, and feature names.
* A `monospace` font presents file and path names.
* Throughout this tutorial, code segments irrelevant to the discussion have been omitted and replaced with ellipses (`...`)

About the author

Nicholas Chase has been involved in Web site development for companies including Lucent Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida. He is the author of three books on Web development, including *Java and XML From Scratch* (Que). He loves to hear from readers and can be reached at *nicholas@nicholaschase.com* .

Section 2. Validation basics

What is validation?

XML files are designed to be easy for people to read and edit. They are also designed for easy data exchange among different systems and different applications. Unfortunately, both of these advantages can work against the need for data to be in a specific format. Validation enables confirmation that XML data follows a specific predetermined structure so that an application can receive it in a predictable way. This structure against which the data is compared can be provided in a number of different ways, including Document Type Definitions (DTDs) and XML schemas.

A document that has been checked against a DTD or schema in this way is considered a "valid" document.

Valid versus well formed

Because *valid* has other meanings in the English language, it is sometimes confused with the XML-specific term *well formed*.

A well-formed document conforms to the rules of XML. All elements have start and end tags, all attributes are enclosed in quotes, all elements are nested correctly, and so on. A document cannot be parsed unless it is well formed.

Just because a document can be parsed, however, does not mean that it is valid in the XML sense. In order to be considered valid, a document must be parsed by a validating parser, which compares it to a predetermined structure.

Valid documents are always well formed, but well-formed documents may not be valid.

Document Type Definitions (DTD)

The concept behind validation actually predates XML itself. When XML was first created it was as an application of SGML. SGML allows different systems to talk to each other by allowing authors to create a DTD. As long as the data followed the DTD, each system could read it.

DTDs define elements that are allowed in a document, what they can contain, and the attributes they can and/or must have.

Compare this simple document and its DTD:

The document:

```
<?xml version="1.0"?>
 <!DOCTYPE memories SYSTEM "memory.dtd">
 <memories>
    <memory tapeid="23412">
       <subdate>5/23/2001</subdate>
       <donor>John Baker</donor>
       <subject>Fishing off Pier 60</subject>
```

```
    </memory>
    <memory tapeid="23692">
       <subdate>8/01/2001</subdate>
       <donor>Elizabeth Davison</donor>
       <subject>Beach volleyball</subject>
    </memory>
 </memories>
```

The DTD:

```
<!ELEMENT memories (memory*)>

 <!ELEMENT memory (subdate, donor, subject)>
 <!ATTLIST memory tapeid CDATA #REQUIRED>

 <!ELEMENT subdate (#PCDATA)>

 <!ELEMENT donor (#PCDATA)>

 <!ELEMENT subject (#PCDATA)>
```

The DTD uses a different syntax from XML, but it describes the various elements and attributes and how they can be used. More information regarding the use and implementation of DTDs is given later in this tutorial. For now, note that the DTD is tied to the XML document via the `DOCTYPE` statement.

XML schemas

DTDs serve their purpose, but serious limitations exist.

For one thing, using XML itself to describe the structure would be much more convenient. For another, there are limitations to what a DTD can specify. No provisions for data types exist, and in some cases, restriction of the order of elements is impossible. These are just some of the limitations developers confront in using DTDs.

XML schemas are one alternative that has been developed to fill some of these holes. Consider the same XML document, this time with a schema document.

The document:

```
<?xml version="1.0"?>
 <memories xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
    <memory tapeid="23412">
       <subdate>5/23/2001</subdate>
       <donor>John Baker</donor>
       <subject>Fishing off Pier 60</subject>
    </memory>
    <memory tapeid="23692">
       <subdate>8/01/2001</subdate>
       <donor>Elizabeth Davison</donor>
       <subject>Beach volleyball</subject>
    </memory>
 </memories>
```

XML Schema:

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

```
<xsd:element name="memories">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="memory" type="memoryType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="memoryType">
  <xsd:sequence>
    <xsd:element name="subdate" type="xsd:date"/>
    <xsd:element name="donor" type="xsd:string"/>
    <xsd:element name="subject" type="xsd:string"/>
  <xsd:attribute name="tapeid" type="idNumber" />
</xsd:complexType>
</xsd:schema>
```

Notice that in this case the syntax for the schema definitions themselves is different from the syntax for DTDs. The syntax for the schema definitions is also the means for tying the definitions into a schema document using XML Namespaces instead of the `DOCTYPE`.

The example document

The complete example XML file for validation in this tutorial consists of information that is part of The Millennium Memory Project, which collects donated home movies and other personal history recordings for posterity.

Each entry consists of a memory and the information about it, such as the donor, location, and subject.

```
<?xml version="1.0"?>
 <memories>
    <memory tapeid="1">
       <media mediaid="1" status="vhs" />
       <subdate>2001-05-23</subdate>
       <donor>John Baker</donor>
       <subject>Fishing with the grandchildren on beautiful day.</subject>
       <location><description>Pier 60</description></location>
    </memory>
    <memory tapeid="2">
       <media mediaid="2" status="vhs"/>
       <subdate>2001-05-18</subdate>
       <donor>Elizabeth Davison</donor>
       <subject>Beach volleyball</subject>
       <location><place>Asbury Park, NJ</place></location>
    </memory>
 </memories>
```

Section 3. Validating a document

How the validation process works

The documents used to describe the structure of a file are simply text files, so no programming is necessary in order to understand them. This tutorial will look at how two Java APIs make use of these files in the validation process. Other programming languages and APIs use the same basic concepts.

A parser must interpret any XML document before an application can use it, typically to create either a DOM document or a SAX stream. In either case, the parser looks at each character of the document and decides whether it is an element, attribute, string of data, and so on.

Validation occurs when the parser also checks the structure of the document against the DTD or schema. However, the parser will only do this if it has been configured to do so. This is normally done through a setting on the parser or the object that creates the parser.

During this time, if problems -- such as an improperly nested element or improperly added attribute -- come up, they must be addressed. This is done through a class designated as the *error handler*.

Create an error handler

An error handler generally extends the `DefaultHandler` helper class, which in turn implements the `ErrorHandler` interface.

The error handler serves one and only one purpose: to deal with irregularities that come up during the parsing of a document.

There are three types of situations:
* Warnings, which generally don't need further action
* Errors, which are actual problems
* Fatal errors (such as a document that is not well formed), which prevent processing from continuing

Each of these situations needs to be dealt with via a method. In this example, the handler simply returns information on the problems encountered, and, on a fatal error, exits.

Compile this class, and it is ready to be referenced by the parser.

```
import org.xml.sax.helpers.DefaultHandler;
 import org.xml.sax.SAXParseException;

 public class ErrorChecker extends DefaultHandler
 {

    public ErrorChecker() {
    }

    public void error (SAXParseException e) {
       System.out.println("Parsing error:  "+e.getMessage());
    }
```

```
    public void warning (SAXParseException e) {
       System.out.println("Parsing problem:  "+e.getMessage());
    }

    public void fatalError (SAXParseException e) {
       System.out.println("Parsing error:  "+e.getMessage());
       System.out.println("Cannot continue.");
       System.exit(1);
    }
 }
```

Validation in JAXP

As discussed in the introduction of this tutorial, it is not necessary to code and run the following examples to understand validation. Should you decide to do so, using JAXP to parse (and ultimately validate) a document involves four steps (The next panel discusses Validation in Xerces Java on page 9):

1. Create the `DocumentBuilderFactory`. Because the `DocumentBuilder`, which actually parses the document, is an interface, it cannot be instantiated directly. Instead, a `DocumentBuilderFactory` is created. This factory has certain properties, such as `isValidating()`, that will determine the behavior of any parsers created with it. To create a validating parser, use `setValidating(true)`.
2. Create the `DocumentBuilder`. Use the `DocumentBuilderFactory` to create the `DocumentBuilder` object, which parses the document.
3. Assign the `ErrorHandler`. It doesn't do any good for the parser to check for problems if it doesn't know what to do with them. Use the `setErrorHandler()` method of the `DocumentBuilder` to tell the parser to send errors to a new `ErrorChecker` object, which was created in the previous panel, Create an error handler on page 7 .
4. Parse the document. If a document is not well formed, `StructureTest` will catch the exception. If it is well formed but there are validation errors, the parser sends them to the `ErrorChecker` object which reports them.

This is the basic principle behind validating a document: create a validating parser, determine the destination for validation errors, and parse the document. The particulars are slightly different for Validation in Xerces Java on page 9 .

```
import javax.xml.parsers.DocumentBuilder;
 import javax.xml.parsers.DocumentBuilderFactory;
 import java.io.File;
 import org.w3c.dom.Document;

 public class StructureTest {

    public static void main (String args[]) {
       File docFile = new File("memory.xml");

       try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          dbf.setValidating(true);

          DocumentBuilder db = dbf.newDocumentBuilder();

          ErrorChecker errors = new ErrorChecker();
          db.setErrorHandler(errors);

          Document doc = db.parse(docFile);

       } catch (Exception e) {
          System.out.print("Parsing problem.");
       }
```

```
   }
}
```

Validation in Xerces Java

Using Xerces Java to validate a document involves the same basic principles as Validation in JAXP on page 8 . Here the steps are:

1. Create the parser. Xerces allows the direct creation of the parser, unlike JAXP, which requires using factories.
2. Turn on validation.
3. Set the error handler.
4. Parse the document.

```
import org.apache.xerces.parsers.DOMParser;
 import java.io.File;
 import org.w3c.dom.Document;

 public class SchemaTest {
    public static void main (String args[]) {
       File docFile = new File("memory.xml");

       try {

          DOMParser parser = new DOMParser();
          parser.setFeature("http://xml.org/sax/features/validation", true);

          parser.setProperty(
 "http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation",
             "memory.xsd");

          ErrorChecker errors = new ErrorChecker();
          parser.setErrorHandler(errors);

          parser.parse("memory.xml");

          } catch (Exception e) {
             System.out.print("Problem parsing the file.");
       }
    }
 }
```

The application directly instantiates a `DOMParser`. Each parser created this way has a set of features, one of which is validation. The parser's `setFeature()` method turns it on.

You can specify the location of the schema document within the XML (as seen in The XML Schema instance document on page 16), or you can specify it within the application itself using a property on the parser, as seen above.

Next assign an error handler. Notice that this is the same class that was used to handle errors in JAXP. The events are the same, so there is no need to code a separate class.

Finally, parse the document. `ErrorChecker` reports any errors.

In this way, all documents are checked against a DTD or schema.

Section 4. Document Type Definitions (DTDs)

External DTDs

In order to validate a document, you must have a standard to validate it against. The oldest and best-supported means for specifying requirements in an XML document is the DTD. A DTD may be internal or external.

When it comes to DTDs, most people are more familiar with the external variety in which the DOCTYPE declaration refers to a file containing the actual definitions.

There are several ways to designate the location of a DTD file. For example, an XHTML file can designate a DTD that determines whether it is following the XHTML Strict, XHTML Transitional, or XHTML Frameset Recommendations developed at the W3C. To designate XHTML Transitional, the author might specify:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

The DOCTYPE declaration consists of several parts:

* <!DOCTYPE Indicates to the processor that this is a DOCTYPE declaration.
* html Indicates the name of the root element for the document. If the document started with anything but <html> it would immediately be deemed invalid.
* PUBLIC A DOCTYPE can designate a publicly recognized DTD, potentially saving the processor a trip to the server to retrieve it. The alternative, SYSTEM, is shown below. A SYSTEM identifier indicates the URI where the DTD can be found.
* "-//W3C//DTD HTML 4.01 Transitional//EN": The actual public identifier for the Transitional XHTML DTD.

For custom DTDs, developers typically use a SYSTEM identifier, such as:

```
<!DOCTYPE memories SYSTEM "http://www.nicholaschase.com/memories.dtd">
```

The parts match those for a PUBLIC identifier, except the declaration shows the location of the DTD.

Typically, the DOCTYPE declaration also specifies the SYSTEM identifier when using a PUBLIC identifier in case the processor doesn't recognize the latter:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
>
```

The external DTD file simply contains the definitions, starting with the Elements on page 11. For an internal DTD, these definitions are part of the XML file itself.

Structure of an internal DTD

An external DTD can specify the contents of many different documents making them somewhat easier to maintain. However, there are times when a valid document needs to stand on its own. In this case, you need to include the DTD information within the document itself.

```
<?xml version="1.0"?>
<!DOCTYPE memories [
   <!ELEMENT memories (memory+) >
   <!ELEMENT memory (#PCDATA) >
]>
<memories>
   <memory>TBD</memory>
   <memory>TBD</memory>
</memories>
```

Ignoring the actual content for a moment, notice the structure of the internal DTD. The DOCTYPE declaration still contains the information but rather than referring to a local or remote file, the actual DTD is included between the brackets.

Elements

In both internal and external DTDs, elements are the foundation of an XML document, so they are typically defined first.

An element definition consists of the ELEMENT keyword, the name of the element, and the content it can contain. The content of an element may be text, other elements, or nothing at all (in the case of empty elements).

```
<!DOCTYPE memories [
    <ELEMENT memories (memory) >
    <ELEMENT memory (subdate, donor, subject, media) >

    <ELEMENT subdate (#PCDATA) >
    <ELEMENT donor (#PCDATA) >
    <ELEMENT subject (#PCDATA) >

    <ELEMENT media EMPTY >
 ]>
```

Designate an element that can contain text with the #PCDATA keyword. This is short for parsed character data; it refers to any text within an element and cannot include markup. Examples are the `subdate`, `donor`, and `subject` elements.

The `memory` and `memories` elements show the syntax used to specify elements that contain only other elements as content.

An element can also be defined as EMPTY, as in the `media` element. Empty elements typically carry all information in attributes. For example:

```
<media type="vhs" />
```

Variable element content

Sometimes an author wants to allow different choices for the content of an element. For example, the data structure contains an element named `location` that may contain either a `place` element or a `description` element. This is written as follows, with a pipe character (|) separating the choices.

```
                    <!ELEMENT location (place | description) >
<!ELEMENT place (#PCDATA) >
<!ELEMENT description (#PCDATA) >
```

Modifiers (*, + and ?)

The DTD that has been created up to this point is very specific. Each element must occur exactly one time, in exactly the same order. The only exception to this is the `location`, where either the `place` or `description` must occur, but not both.

Modifiers offer more flexibility in design. They are:

* * : An element designated with the * modifier may appear 0 or more times.
* + : An element designated with the + modifier must appear 1 or more times.
* ? : An element designated with the ? modifier must appear 0 or 1 time.

The code below shows the DTD modified so the number of `memory` elements is unrestricted. It also shows that there must be at least one `subject`, but there can be more than one. Finally, the `donor` name is not required, but can appear only once if it is present. All unmodified elements must appear once and only once.

```
<!DOCTYPE memories [
    <!ELEMENT memories (memory)* >
    <!ELEMENT memory (media,
                      subdate,
                      donor?,
                      subject+,
                      location) >
    <!ELEMENT subdate (#PCDATA) >
    <!ELEMENT donor (#PCDATA) >
    <!ELEMENT subject (#PCDATA) >
    <!ELEMENT location (place |
                        description) >
    <!ELEMENT description (#PCDATA) >
    <!ELEMENT place (#PCDATA) >
    <!ELEMENT media EMPTY >
 ]>
```

Note that these restrictions are *per element*. The `donor` can only appear once within a `memory` element, but can appear in every instance of `memory`.

Ordering child elements

XML is primarily a hierarchical system, consisting of elements within elements within other elements. These relationships are known as *parent-child* relationships. For example, the `media` element is contained within the `memory` element, so the `memory` element is considered the *parent* of the `media` element. Conversely, the `media` element is the *child* of the `memory` element. One parent, such as `memory`, may have multiple children.

The order of child elements can also be determined by looking at a DTD. Paradoxically, while child elements must always appear in the order in which they appear in the DTD, the DTD can be written in such a way that the children can appear in any order.

Strictly speaking, the required order doesn't change, but the options do. For example, the current DTD specifies that the `location` element can have either a `place` or a `description`.

```
<!ELEMENT location (place|description) >
```

If this choice could be repeated, as in:

```
<!ELEMENT location (place|description)* >
```

then the `location` could contain a `place` and a `description`, in any order. The same thought could be applied to the `memory` element:

```
<!ELEMENT memory (media | subdate | donor?| subject+| location)* >
```

In this case, the elements can appear in any order because the DTD allows unlimited choices. First a `subdate` could be chosen, then a `location`, then a `donor`, and so on. Notice, however, that once this technique is employed, certain previous restrictions become useless. Because choices can be made more than once, any of the specified elements can be chosen any number of times, or not at all.

This is a serious limitation of DTDs. It is overcome through the use of XML schemas, which allow much greater control. Schemas are also useful when defining *mixed content*.

Mixed content

A mixed content element contains both text and other elements. One good example of this is text containing HTML markup. Consider the following potential `subject`:

```
<subject>
    A reading of Charles Dickens' <i>A Christmas Carol</i>. Absolutely marvelous!
 </subject>
```

This is known as mixed content because it has both character data and an element >). In order to make this acceptable to a validating parser, the `i` element must be defined and the `subject` element must be allowed to take any number of either `#PCDATA` or `i` choices. To allow common markup, the DTD needs to read:

```
<!ELEMENT i (#PCDATA) >
 <!ELEMENT b (#PCDATA) >
 <!ELEMENT h1 (#PCDATA) >
 <!ELEMENT br EMPTY >
 <!ELEMENT p (#PCDATA) >
 <!ELEMENT subject (#PCDATA|i|b|h1|br|p)* >
```

Note that while this does solve the problem, there is no way to constrain the order. This, too, is a problem solved by XML Schema.

Define attributes

While it is possible to create an XML structure with nothing but elements, the more common situation is elements with attributes. Attributes must also be defined if they are to appear on elements in a validated document.

There are several ways to define an attribute. The first is to simply designate it as character data, or CDATA:

```
<!ATTLIST memory tapeid CDATA #REQUIRED >
```

In this case, the DTD assigns the attribute `tapeid` to the `memory` element. The `tapeid`

attribute consists of character data, and is required. An element can also be designated as #IMPLIED or #FIXED, in which case a default value must also be specified.

Some attributes are enumerated, meaning that a value must be chosen from a predetermined list. For example:

```
<!ATTLIST media type (8mm | vhsc | vhs | audio) '8mm' #IMPLIED >
```

In this case, the document must choose a value from the list. If no value is provided, the parser will use the default value of 8mm. This is the case in any document for which a DTD is present, even if the parser is not validating.

Multiple attributes can be designated with a single ATTLIST definition:

```
<!ATTLIST media type (8mm | vhsc | vhs | audio) '8mm' #IMPLIED
                 length CDATA >
```

A second means for defining attribute content involves IDs and IDREFs.

IDs and IDREFs

It is sometimes necessary to "link" data together with the use of an *identifier*, much the way primary and foreign keys work in a relational database. For example, it might be a requirement that the memory identifier matches up with the media identifier, so that a memory can be located. ID and IDREF datatypes allow you to enforce such data integrity:

```
<!ATTLIST media mediaid ID #REQUIRED >
 <!ATTLIST memory tapeid IDREF #REQUIRED >
```

These notations add two restrictions to the file. First, the value of each mediaid must be unique, and second, the value of each tapeid must match an existing mediaid.

Each element can have a maximum of one ID attribute, but there is no way to force an IDREF to reference a particular ID.

The complete DTD and document

The DTD:

```
<!ELEMENT memories (memory)* >
 <!ELEMENT memory (media | subdate | donor?| subject+| location)* >
 <!ATTLIST memory tapeid IDREF #REQUIRED >
 <!ELEMENT subdate (#PCDATA) >
 <!ELEMENT donor (#PCDATA) >
 <!ELEMENT subject (#PCDATA) >
 <!ELEMENT location (place|description) >
 <!ELEMENT description (#PCDATA) >
 <!ELEMENT place (#PCDATA) >
 <!ELEMENT media EMPTY >
 <!ATTLIST media mediaid ID #REQUIRED >
```

The document:

```
<?xml version="1.0"?>
 <!DOCTYPE memories SYSTEM "memory.dtd">
```

```
<memories>
   <memory tapeid="1">
      <media mediaid="1" status="vhs" />
      <subdate>2001-05-23</subdate>
      <donor>John Baker</donor>
      <subject>Fishing off Pier 60</subject>
      <location><description>Outside in the woods</description></location>
   </memory>
   <memory tapeid="2">
         <media mediaid="2" status="vhs"/>
      <subdate>2001-05-18</subdate>
      <donor>Elizabeth Davison</donor>
      <subject>Beach volleyball</subject>
      <location><place>Clearwater beach</place></location>
   </memory>
</memories>
```

DTD limitations

DTDs are extremely limited when it comes to the actual types of data they can include. For example, there is no way to constrain data to be numeric, or dates. Attributes can be constrained as unique `ID`s, but there is no way to determine what their datatype should be. (And oddly, an ID can't be a number, contrary to common practice in relational databases!)

There are also limitations on Ordering child elements on page 12 . In addition, there cannot be more than one element with a particular name, so there is no way to create a different definition for an element that might appear in more than one context (such as the children of two different parents).

All of these difficulties are resolved with the use of XML Schema.

Section 5. XML Schema

The XML Schema instance document

In contrast to DTDs, schema documents are built in XML itself. Validation using schemas requires two documents: the schema document, and the instance document.

The schema document is the document containing the structure, and the instance document is the document containing the actual XML data. An application determines the schema for an instance document in one of two ways:

1. From the document itself: While documents use the DOCTYPE declaration to point to an external DTD, they use attributes and namespaces to point to an external schema document:

   ```
   <?xml version="1.0"?>
    <memories xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xsi:noNamespaceSchemaLocation='memory.xsd'>
       <memory tapeid="idnum">
    ...
   ```

   First create the namespace itself, then use the `noNamespaceSchemaLocation` attribute to determine the location. Schemas can also be created for a particular target namespace. In that case, specify the `targetNamespace` in the schema document itself.

2. Through properties set within the application: With Xerces, set the `http://apache.org/xml/properties/schema/external-schemaLocation` and the `http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation` properties to determine the location of the schema document, as seen in Validation in Xerces Java on page 9 .

Structure of a schema document

A schema document is simply an XML document with predefined elements and attributes describing the structure of another XML document.

Consider this sample schema document:

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

 <xsd:element name="memories">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="memory" maxOccurs="unbounded" type="memoryType"/>
       </xsd:sequence>
    </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="memoryType">
    <xsd:sequence>
       <xsd:element name="media">
          <xsd:complexType>
             <xsd:attribute name="mediaid" type="xsd:string" />
             <xsd:attribute name="status" type="xsd:string" />
          </xsd:complexType>
```

```
      </xsd:element>
      <xsd:element name="subdate" type="xsd:string"/>
      <xsd:element name="donor" type="xsd:string"/>
      <xsd:element name="subject" type="xsd:string"/>
      <xsd:element name="location" type="locationType" />
   </xsd:sequence>
   <xsd:attribute name="tapeid" type="xsd:string" />
   <xsd:attribute name="status" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="locationType">
   <xsd:choice>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="place" type="xsd:string" />
   </xsd:choice>
</xsd:complexType>
</xsd:schema>
```

This document, the XML Schema equivalent of the DTD built earlier in the tutorial, shows some of the structures used to define the content of an XML document. The development of a schema starts with the definition of elements.

Elements and built-in types

All XML documents are built on elements. Defining an element in a schema document is a matter of naming it and assigning it a type. This type designation can reference a custom type, or one of the built-in types listed in the XML Schema Recommendation. This example shows built-in types:

```
<xsd:element name="subdate" type="xsd:date"/>
 <xsd:element name="donor" type="xsd:string"/>
 <xsd:element name="subject" type="xsd:string"/>
 <xsd:element name="description" type="xsd:string" />
 <xsd:element name="place" type="xsd:string" />
```

These are simple elements that contain only text. The `subdate` element has been further constrained, however, to dates, in the format of yyyy-mm-dd, as defined in the W3C XML Schema Recommendation.

Forty two simple types are defined as part of the recommendation, including `string`, `int`, `date`, `decimal`, `boolean`, `timeDuration`, and `uriReference`.

You can also create new types.

Simple types: restricting values

In addition to the built-in simple types, a schema can designate the creation of new simple types. These types can range from text in particular formats, such as a phone number or product number, to a numeric range, to an enumerated list.

The example project needs two new simple types. The first is the `idNumber`:

```
<xsd:simpleType name="idNumber">
    <xsd:restriction  base="xsd:integer">
       <xsd:minInclusive value="1" />
       <xsd:maxInclusive value="100000" />
    </xsd:restriction>
```

```
 </xsd:simpleType>
```

This type, used for the `tapeid` attribute, simply limits the values to an integer between 1 and 100000.

Simple types: enumeration

You can use another type of derivation by restriction to create an enumerated list, such as this one to limit the values for the media type:

```
<xsd:simpleType name="mediaType">
    <xsd:restriction  base="xsd:string">
       <xsd:enumeration value="8mm" />
       <xsd:enumeration value="vhs" />
       <xsd:enumeration value="vhsc" />
       <xsd:enumeration value="digital" />
       <xsd:enumeration value="audio" />
    </xsd:restriction>
 </xsd:simpleType>
```

You can use derived simple types in the same way as the built-in types.

Complex types: attributes

One of the restrictions of a simple type is that these elements cannot contain attributes. In order to add attributes to an element, you must convert it to a `complexType`.

One way to do this is through the use of *anonymous* . This involves adding a `complexType` element as a child to the `element` element.

```
<xsd:element name="media">
    <xsd:complexType>
       <xsd:attribute name="mediaid" type="xsd:integer" />
       <xsd:attribute name="status" type="mediaType" />
    </xsd:complexType>
 </xsd:element>
```

In this case, the `media` element now has two attributes, including one that follows the enumerated `mediaType`.

You can also create and name complex types.

Complex types: elements

Adding children to an element also requires the use of complex types. To simply list one or more child elements, use the `sequence` element:

```
<xsd:element name="memories">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="memory" type="memoryType"/>
       </xsd:sequence>
    </xsd:complexType>
```

```
 </xsd:element>
```

This example shows only one child element, but you still need the `sequence` element. The type `memoryType` combines many of the techniques seen so far:

```
<xsd:complexType name="memoryType">
    <xsd:sequence>
       <xsd:element name="media">
          <xsd:complexType>
             <xsd:attribute name="mediaid" type="xsd:integer" />
             <xsd:attribute name="status" type="mediaType" />
          </xsd:complexType>
       </xsd:element>
       <xsd:element name="subdate" type="xsd:date"/>
       <xsd:element name="donor" type="xsd:string"/>
       <xsd:element name="subject" type="xsd:string"/>
       <xsd:element name="location" type="locationType" />
    </xsd:sequence>
    <xsd:attribute name="tapeid" type="idNumber" />
    <xsd:attribute name="status" type="xsd:string" />
 </xsd:complexType>
```

The `memoryType` type also includes a reference to the `locationType`, which allows the user to choose between potential children for an element.

Element choices

The `sequence` element shows all of the possible children of an element. In some cases, however, you want to choose only one element from a list of alternatives. For that, you need the `choice` element:

```
<xsd:complexType name="locationType">
    <xsd:choice>
       <xsd:element name="description" type="xsd:string" />
       <xsd:element name="place" type="xsd:string" />
    </xsd:choice>
 </xsd:complexType>
```

Either the `description` or the `place` element may appear as a child of any element of `locationType`, but not both. Use this technique to define a choice of attributes as well as elements.

Optional and repeated elements

At this point all elements and attributes added to the schema must appear exactly once. This is obviously unacceptable in most cases. Using `minOccurs` and `maxOccurs`, you can control whether or not an item must appear, and whether or not it may repeat. In this example, the schema requires a `subject`, and allows it to appear as many as five times within a single element:

```
...
 <xsd:element name="subject" minOccurs="1" maxOccurs="5" type="xsd:string"/>
 ...
```

Sometimes you don't want an upper limit. For example, the `memory` element may be specified as optional, but if present, it may appear an unlimited number of times in the

`memories` element:

```
<xsd:element name="memories">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="memory"
                  minOccurs="0" maxOccurs="unbounded"
                  type="memoryType"/>
       </xsd:sequence>
    </xsd:complexType>
 </xsd:element>
```

This capability also comes in handy when dealing with Mixed content on page 20.

Mixed content

There are times when an element contains mixed content, as opposed to just elements, or just text. For example, the `subject` element might contain HTML markup:

```
<subject>
    This lab specializes in studying drosophila, the common fruit fly.
    They also bred the genetic variation ether-a-go-go.
 </subject>
```

This content couldn't be described as `xsd:string` because it contains elements. Similarly, just listing the `i` and b elements in a sequence wouldn't do because they contain text. Instead, the `subject` element must be defined as `mixed`:

```
<xsd:element name="subject">
   <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:element name="i" minOccurs="0" maxOccurs="unbounded" type="xsd:string" />
        <xsd:element name="b" minOccurs="0" maxOccurs="unbounded" type="xsd:string" />
      </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
```

Note that including HTML is not the only time that mixed content comes into play. Often `system` will contain narrative or other data that includes items such as names, addresses, acronyms, or terms, that are set off as elements so they can be found or styled appropriately.

This method of creating mixed content is an improvement over DTDs because it allows better control over the number and order of elements. Of course, there will be times when you don't want that either.

Unconstrained order

Occasionally the general content of an element needs to be constrained, but not the order in which it appears. This is particularly true for Mixed content on page 20 . To create an element that does not constrain the order of its children, use the `all` element instead of `sequence`.

```
<xsd:element name="subject">
   <xsd:complexType mixed="true">
      <xsd:all>
```

```
       <xsd:element name="i" minOccurs="0" maxOccurs="1"  type="xsd:string" />
       <xsd:element name="b" minOccurs="0" maxOccurs="1"  type="xsd:string" />
     </xsd:all>
  </xsd:complexType>
 </xsd:element>
```

Notice that `maxOccurs` has been set to `1`, rather than `unbounded`. This is a requirement for using the `all` element. The `minOccurs` and `maxOccurs` attributes must be either `0` or `1`. Combining and nesting these groups can create an element where the types of elements, but not their order or frequency, is constrained.

Schema documents also support completely unconstrained elements.

Completely unconstrained elements

The ultimate in an unconstrained element would be one for which any content whatsoever is permitted. You can achieve this via the built-in `anyType` type. For example, you can define the `location` element to contain any content, in any order, with any frequency. This includes not only text, but also elements:

```
<xsd:element name="description" type="xsd:anyType" />
```

Combining all of these techniques leads to the complete document.

The complete schema and instance documents

The XML Schema document:

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

 <xsd:element name="memories">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name="memory" minOccurs = "0" maxOccurs="unbounded" type="memoryType"/>
       </xsd:sequence>
    </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="memoryType">
    <xsd:sequence>
       <xsd:element name="media">
          <xsd:complexType>
             <xsd:attribute name="mediaid" type="xsd:integer" />
             <xsd:attribute name="status" type="mediaType" />
          </xsd:complexType>
       </xsd:element>
       <xsd:element name="subdate" type="xsd:date"/>
       <xsd:element name="donor" type="xsd:string"/>
       <xsd:element name="subject">
          <xsd:complexType mixed="true">
              <xsd:all>
                 <xsd:element name="i" minOccurs="0" maxOccurs="1" type="xsd:string" />
                 <xsd:element name="b" minOccurs="0" maxOccurs="1" type="xsd:string" />
              </xsd:all>
          </xsd:complexType>
       </xsd:element>
       <xsd:element name="location" type="locationType" />
    </xsd:sequence>
    <xsd:attribute name="tapeid" type="idNumber" />
    <xsd:attribute name="status" type="xsd:string" />
```

```
 </xsd:complexType>

 <xsd:complexType name="locationType">
    <xsd:choice>
       <xsd:element name="description" type="xsd:anyType" />
       <xsd:element name="place" type="xsd:string" />
    </xsd:choice>
 </xsd:complexType>

 <xsd:simpleType name="idNumber">
    <xsd:restriction  base="xsd:integer">
       <xsd:minInclusive value="1" />
       <xsd:maxInclusive value="100000" />
    </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="mediaType">
    <xsd:restriction  base="xsd:string">
       <xsd:enumeration value="8mm" />
       <xsd:enumeration value="vhs" />
       <xsd:enumeration value="vhsc" />
       <xsd:enumeration value="digital" />
       <xsd:enumeration value="audio" />
    </xsd:restriction>
 </xsd:simpleType>
 </xsd:schema>
```

The instance document:

```
<?xml version="1.0"?>
 <memories xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xsi:noNamespaceSchemaLocation='memory.xsd'>
    <memory tapeid="1">
       <media mediaid="1" status="vhs" />
       <subdate>2001-05-23</subdate>
       <donor>John Baker</donor>
       <subject>Fishing off Pier 60</subject>
       <location>
          <description>Outside in the woods</description>
       </location>
    </memory>
    <memory tapeid="2">
       <media mediaid="2" status="vhs"/>
       <subdate>2001-05-18</subdate>
       <donor>Elizabeth Davison</donor>
       <subject>Beach volleyball</subject>
       <location>
          <place>Clearwater beach</place>
       </location>
    </memory>
 </memories>
```

Section 6. Validation summary

Summary

This tutorial showed you how to create both DTD and XML Schema documents to validate your XML documents against. The strengths of schemas over DTDs were discussed.

The tutorial also discussed validation from the perspective of two different Java APIs (JAXP and Xerces), but remember that XML is a platform-independent means for representing information. Validating parsers are available in C++, Perl, and other languages, and the concepts are the same.

Resources

There is a lot of good information on validating XML documents on the Web.

* For a basic grounding in XML read through the *Introduction to XML* tutorial.
* Read the W3C's original *XML 1.0 Recommendation* to see the definition of Document Type Definitions (DTDs).
* Read the *XML Schema Recommendation* at the W3C site. It also includes an XML Schema Primer explaining the different options available to you.
* Read Kevin Williams' discussion of *Why XML Schema beats DTDs hands-down for data* .
* Visit XML.org's registry of over 500 *industry standard DTDs* .
* Visit XML.org's registry of *industry standard schemas* .
* Order *XML and Java from Scratch* , by Nicholas Chase, the author of this tutorial.
* Download IBM's *Schema Quality Checker* from alphaWorks (free download with 90-day license, renewable). It evaluates your schema documents for errors.
* Download *the Java 2 SDK* , Standard Edition version 1.3.1.
* Download *JAXP 1.1* , the JavaTM APIs for XML Processing.
* Download the *the Xerces parser for Java* from the Apache XML project.
* Download the *the Xerces parser for C++* from the Apache XML project.
* Download the *the Xerces parser for Perl* from the Apache XML project.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.