Introduction to XML

Doug Tidwell (dtidwell@us.ibm.com)
XML Evangelist
IBM

07 August 2002

XML, the Extensible Markup Language, has gone from the latest buzzword to an entrenched eBusiness technology in record time. This newly revised tutorial discusses what XML is, why it was developed, and how it's shaping the future of electronic commerce. It also covers a variety of important XML programming interfaces and standards, and ends with two case studies showing how companies are using XML to solve business problems.

About this tutorial

Should I take this tutorial?

This newly revised tutorial discusses what XML is, why it was developed, and how it's shaping the future of electronic commerce. Along the way, it also takes a look at several XML standards and programming interfaces, shows how you can get started with this technology, and describes how a couple of companies have built XML-based solutions to simplify and streamline their enterprises.

In this tutorial, you'll learn:

- Why XML was created
- The rules of XML documents
- How to define what an XML document can and cannot contain
- Programming interfaces that work with XML documents
- What the main XML standards are and how they work together
- How companies are using XML in the real world

What is XML?

Introduction

XML, or Extensible Markup Language, is a markup language that you can use to create your own tags. It was created by the World Wide Web Consortium (W3C) to overcome the limitations of HTML, the Hypertext Markup Language that is the basis for all Web pages. Like HTML, XML is based on SGML -- Standard Generalized Markup Language. Although SGML has been used in the publishing industry for decades, its perceived complexity intimidated many people that otherwise

might have used it (SGML also stands for "Sounds great, maybe later"). XML was designed with the Web in mind.

Why do we need XML?

HTML is the most successful markup language of all time. You can view the simplest HTML tags on virtually any device, from palmtops to mainframes, and you can even convert HTML markup into voice and other formats with the right tools. Given the success of HTML, why did the W3C create XML? To answer that question, take a look at this document:

```
<p><b>Mrs. Mary McGoon</b>
<br>
1401 Main Street
<br>
Anytown, NC 34829</p>
```

The trouble with HTML is that it was designed with humans in mind. Even without viewing the above HTML document in a browser, you and I can figure out that it is someone's postal address. (Specifically, it's a postal address for someone in the United States; even if you're not familiar with all the components of U.S. postal addresses, you could probably guess what this represents.)

As humans, you and I have the intelligence to understand the meaning and intent of most documents. A machine, unfortunately, can't do that. While the tags in this document tell a browser how to display this information, the tags don't tell the browser **what the information is**. You and I know it's an address, but a machine doesn't.

Rendering HTML

To render HTML, the browser merely follows the instructions in the HTML document. The paragraph tag tells the browser to start rendering on a new line, typically with a blank line beforehand, while the two break tags tell the browser to advance to the next line without a blank line in between. While the browser formats the document beautifully, the machine still doesn't know this is an address.

Figure 1. HTML address

Processing HTML

To wrap up this discussion of the sample HTML document, consider the task of extracting the postal code from this address. Here's an (intentionally brittle) algorithm for finding the postal code in HTML markup:

If you find a paragraph with two `<br>` tags, the postal code is the second word after the first comma in the second break tag.

Although this algorithm works with this example, there are any number of perfectly valid addresses worldwide for which this simply wouldn't work. Even if you could write an algorithm that found the postal code for any address written in HTML, there are any number of paragraphs with two break tags that don't contain addresses at all. Writing an algorithm that looks at any HTML paragraph and finds any postal codes inside it would be extremely difficult, if not impossible.

A sample XML document

Now let's look at a sample XML document. With XML, you can assign some meaning to the tags in the document. More importantly, it's easy for a machine to process the information as well. You can extract the postal code from this document by simply locating the content surrounded by the `<postal-code>` and `</postal-code>` tags, technically known as the `<postal-code>` *element*.

```
<address>
  <name>
    <title>Mrs.</title>
    <first-name>
      Mary
    </first-name>
    <last-name>
      McGoon
    </last-name>
  </name>
  <street>
    1401 Main Street
  </street>
  <city>Anytown</city>
  <state>NC</state>
  <postal-code>
    34829
  </postal-code>
</address>
```

Tags, elements, and attributes

There are three common terms used to describe parts of an XML document: *tags*, *elements*, and *attributes*. Here is a sample document that illustrates the terms:

```
<address>
  <name>
    <title>Mrs.</title>
    <first-name>
      Mary
    </first-name>
    <last-name>
      McGoon
    </last-name>
  </name>
  <street>
    1401 Main Street
  </street>
  <city state="NC">Anytown</city>
  <postal-code>
    34829
  </postal-code>
</address>
```

- A tag is the text between the left angle bracket (`<`) and the right angle bracket (`>`). There are starting tags (such as `<name>`) and ending tags (such as `</name>`)
- An element is the starting tag, the ending tag, and everything in between. In the sample above, the `<name>` element contains three child elements: `<title>`, `<first-name>`, and `<last-name>`.
- An attribute is a name-value pair inside the starting tag of an element. In this example, `state` is an attribute of the `<city>` element; in earlier examples, `<state>` was an element (see A sample XML document).

How XML is changing the Web

Now that you've seen how developers can use XML to create documents with self-describing data, let's look at how people are using those documents to improve the Web. Here are a few key areas:

- **XML simplifies data interchange.** Because different organizations (or even different parts of the same organization) rarely standardize on a single set of tools, it can take a significant amount of work for applications to communicate. Using XML, each group creates a single utility that transforms their internal data formats into XML and vice versa. Best of all, there's a good chance that their software vendors already provide tools to transform their database records (or LDAP directories, or purchase orders, and so forth) to and from XML.
- **XML enables smart code.** Because XML documents can be structured to identify every important piece of information (as well as the relationships between the pieces), it's possible to write code that can process those XML documents without human intervention. The fact that software vendors have spent massive amounts of time and money building XML development tools means writing that code is a relatively simple process.
- **XML enables smart searches.** Although search engines have improved steadily over the years, it's still quite common to get erroneous results from a search. If you're searching HTML pages for someone named "Chip," you might also find pages on chocolate chips, computer chips, wood chips, and lots of other useless matches. Searching XML documents for `<first-name>` elements that contained the text `Chip` would give you a much better set of results.

I'll also discuss real-world uses of XML in Case studies .

XML document rules

Overview: XML document rules

If you've looked at HTML documents, you're familiar with the basic concepts of using tags to mark up the text of a document. This section discusses the differences between HTML documents and XML documents. It goes over the basic rules of XML documents, and discusses the terminology used to describe them.

One important point about XML documents: **The XML specification requires a parser to reject any XML document that doesn't follow the basic rules.** Most HTML parsers will accept sloppy markup, making a guess as to what the writer of the document intended. To avoid the loosely

structured mess found in the average HTML document, the creators of XML decided to enforce document structure from the beginning.

(By the way, if you're not familiar with the term, a *parser* is a piece of code that attempts to read a document and interpret its contents.)

Invalid, valid, and well-formed documents

There are three kinds of XML documents:

- **Invalid documents** don't follow the syntax rules defined by the XML specification. If a developer has defined rules for what the document can contain in a DTD or schema, and the document doesn't follow those rules, that document is invalid as well. (See Defining document content for a proper introduction to DTDs and schemas for XML documents.)
- **Valid documents** follow both the XML syntax rules and the rules defined in their DTD or schema.
- **Well-formed documents** follow the XML syntax rules but don't have a DTD or schema.

The root element

An XML document must be contained in a single element. That single element is called the **root element**, and it contains all the text and any other elements in the document. In the following example, the XML document is contained in a single element, the `<greeting>` element. Notice that the document has a comment that's outside the root element; that's perfectly legal.

```
<?xml version="1.0"?>
<!-- A well-formed document -->
<greeting>
  Hello, World!
</greeting>
```

Here's a document that doesn't contain a single root element:

```
<?xml version="1.0"?>
<!-- An invalid document -->
<greeting>
  Hello, World!
</greeting>
<greeting>
  Hola, el Mundo!
</greeting>
```

An XML parser is required to reject this document, regardless of the information it might contain.

Elements can't overlap

XML elements can't overlap. Here's some markup that isn't legal:

```
<!-- NOT legal XML markup -->
<p>
  <b>I <i>really
  love</b> XML.
  </i>
</p>
```

If you begin a `<i>` element inside a `<b>` element, you have to end it there as well. If you want the text `XML` to appear in italics, you need to add a second `<i>` element to correct the markup:

```
<!-- legal XML markup -->
<p>
  <b>I <i>really
  love</i></b>
  <i>XML.</i>
</p>
```

An XML parser will accept only this markup; the HTML parsers in most Web browsers will accept both.

End tags are required

You can't leave out any end tags. In the first example below, the markup is not legal because there are no end paragraph (`</p>`) tags. While this is acceptable in HTML (and, in some cases, SGML), an XML parser will reject it.

```
<!-- NOT legal XML markup -->
<p>Yada yada yada...
<p>Yada yada yada...
<p>...
```

If an element contains no markup at all it is called an **empty element**; the HTML break (`<br>`) and image (`<img>`) elements are two examples. In empty elements in XML documents, you can put the closing slash in the start tag. The two break elements and the two image elements below mean the same thing to an XML parser:

```
<!-- Two equivalent break elements -->
<br></br>
<br />

<!-- Two equivalent image elements -->
<img src="../img/c.gif"></img>
<img src="../img/c.gif" />
```

Elements are case sensitive

XML elements are case sensitive. In HTML, `<h1>` and `<H1>` are the same; in XML, they're not. If you try to end an `<h1>` element with a `</H1>` tag, you'll get an error. In the example below, the heading at the top is illegal, while the one at the bottom is fine.

```
<!-- NOT legal XML markup -->
<h1>Elements are
  case sensitive</H1>

<!-- legal XML markup -->
<h1>Elements are
  case sensitive</h1>
```

Attributes must have quoted values

There are two rules for attributes in XML documents:

- Attributes must have values
- Those values must be enclosed within quotation marks

Compare the two examples below. The markup at the top is legal in HTML, but not in XML. To do the equivalent in XML, you have to give the attribute a value, and you have to enclose it in quotes.

```
<!-- NOT legal XML markup -->
<ol compact>

<!-- legal XML markup -->
<ol compact="yes">
```

You can use either single or double quotes, just as long as you're consistent.

If the value of the attribute contains a single or double quote, you can use the other kind of quote to surround the value (as in `name="Doug's car"`), or use the entities `&quot;` for a double quote and `&apos;` for a single quote. An *entity* is a symbol, such as `&quot;`, that the XML parser replaces with other text, such as `"`.

XML declarations

Most XML documents start with an *XML declaration* that provides basic information about the document to the parser. An XML declaration is recommended, but not required. If there is one, it must be the first thing in the document.

The declaration can contain up to three name-value pairs (many people call them attributes, although technically they're not). The `version` is the version of XML used; currently this value must be `1.0`. The `encoding` is the character set used in this document. The `ISO-8859-1` character set referenced in this declaration includes all of the characters used by most Western European languages. If no `encoding` is specified, the XML parser assumes that the characters are in the `UTF-8` set, a Unicode standard that supports virtually every character and ideograph from the world's languages.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
```

Finally, `standalone,` which can be either `yes` or `no`, defines whether this document can be processed without reading any other files. For example, if the XML document doesn't reference any other files, you would specify `standalone="yes"`. If the XML document references other files that describe what the document can contain (more about those files in a minute), you could specify `standalone="no"`. Because `standalone="no"` is the default, you rarely see `standalone` in XML declarations.

Other things in XML documents

There are a few other things you might find in an XML document:

- **Comments:** Comments can appear anywhere in the document; they can even appear before or after the root element. A comment begins with `<!--` and ends with `-->`. A comment can't contain a double hyphen (`--`) except at the end; with that exception, a comment can contain

anything. Most importantly, any markup inside a comment is ignored; if you want to remove a large section of an XML document, simply wrap that section in a comment. (To restore the commented-out section, simply remove the comment tags.) Here's some markup that contains a comment:

```
<!-- Here's a PI for Cocoon: -->
<?cocoon-process type="sql"?>
```

- **Processing instructions:** A processing instruction is markup intended for a particular piece of code. In the example above, there's a processing instruction (sometimes called a PI) for Cocoon, an XML processing framework from the Apache Software Foundation. When Cocoon is processing an XML document, it looks for processing instructions that begin with `cocoon-process`, then processes the XML document accordingly. In this example, the `type="sql"` attribute tells Cocoon that the XML document contains a SQL statement.

```
<!-- Here's an entity: -->
<!ENTITY dw "developerWorks">
```

- **Entities:** The example above defines an *entity* for the document. Anywhere the XML processor finds the string `&dw;`, it replaces the entity with the string `developerWorks`. The XML spec also defines five entities you can use in place of various special characters. The entities are:
  - `<` for the less-than sign
  - `>` for the greater-than sign
  - `&quot;` for a double-quote
  - `&apos;` for a single quote (or apostrophe)
  - `&` for an ampersand.

Namespaces

XML's power comes from its flexibility, the fact that you and I and millions of other people can define our own tags to describe our data. Remember the sample XML document for a person's name and address? That document includes the `<title>` element for a person's courtesy title, a perfectly reasonable choice for an element name. If you run an online bookstore, you might create a `<title>` element for the title of a book. If you run an online mortgage company, you might create a `<title>` element for the title to a piece of property. All of those are reasonable choices, but all of them create elements with the same name. How do you tell if a given `<title>` element refers to a person, a book, or a piece of property? With *namespaces*.

To use a namespace, you define a *namespace prefix* and map it to a particular string. Here's how you might define namespace prefixes for our three `<title>` elements:

```
<?xml version="1.0"?>
<customer_summary
  xmlns:addr="http://www.xyz.com/addresses/"
  xmlns:books="http://www.zyx.com/books/"
  xmlns:mortgage="http://www.yyz.com/title/"
>
... <addr:name><title>Mrs.</title> ... </addr:name> ...
... <books:title>Lord of the Rings</books:title> ...
... <mortgage:title>NC2948-388-1983</mortgage:title> ...
```

In this example, the three namespace prefixes are `addr`, `books`, and `mortgage`. Notice that defining a namespace for a particular element means that all of its child elements belong to the same namespace. The first `<title>` element belongs to the `addr` namespace because its parent element, `<addr:Name>`, does.

One final point: **The string in a namespace definition is just a string.** Yes, these strings look like URLs, but they're not. You could define `xmlns:addr="mike"` and that would work just as well. The only thing that's important about the namespace string is that it's unique; that's why most namespace definitions look like URLs. The XML parser does not go to `http://www.zyx.com/books/` to search for a DTD or schema, it simply uses that text as a string. It's confusing, but that's how namespaces work.

Defining document content

Overview: Defining document content

So far in this tutorial you've learned about the basic rules of XML documents; that's all well and good, but you need to define the elements you're going to use to represent data. You'll learn about two ways of doing that in this section.

- One method is to use a **Document Type Definition,** or **DTD.** A DTD defines the elements that can appear in an XML document, the order in which they can appear, how they can be nested inside each other, and other basic details of XML document structure. DTDs are part of the original XML specification and are very similar to SGML DTDs.
- The other method is to use an **XML Schema.** A schema can define all of the document structures that you can put in a DTD, and it can also define data types and more complicated rules than a DTD can. The W3C developed the XML Schema specification a couple of years after the original XML spec.

Document Type Definitions

A DTD allows you to specify the basic structure of an XML document. The next couple of sections look at fragments of DTDs. First of all, here's a DTD that defines the basic structure of the address document example in the section, What is XML? :

```
<!-- address.dtd -->
<!ELEMENT address (name, street, city, state, postal-code)>
<!ELEMENT name (title? first-name, last-name)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>
```

This DTD defines all of the elements used in the sample document. It defines three basic things:

- An `<address>` element contains a `<name>`, a `<street>`, a `<city>`, a `<state>`, and a `<postal-code>`. All of those elements must appear, and they must appear in that order.
- A `<name>` element contains an optional `<title>` element (the question mark means the title is optional), followed by a `<first-name>` and a `<last-name>` element.

- All of the other elements contain text. (`#PCDATA` stands for parsed character data; you can't include another element in these elements.)

Although the DTD is pretty simple, it makes it clear what combinations of elements are legal. An address document that has a `<postal-code>` element before the `<state>` element isn't legal, and neither is one that has no `<last-name>` element.

Also, **notice that DTD syntax is different from ordinary XML syntax.** (XML Schema documents, by contrast, are themselves XML, which has some interesting consequences.) Despite the different syntax for DTDs, you can still put an ordinary comment in the DTD itself.

Symbols in DTDs

There are a few symbols used in DTDs to indicate how often (or whether) something may appear in an XML document. Here are some examples, along with their meanings:

- `<!ELEMENT address (name, city, state)>`
  The `<address>` element must contain a `<name>`, a `<city>`, and a `<state>` element, in that order. All of the elements are required. **The comma indicates a list of items.**
- `<!ELEMENT name (title?, first-name, last-name)>`
  This means that the `<name>` element contains an optional `<title>` element, followed by a mandatory `<first-name>` and a `<last-name>` element. **The question mark indicates that an item is optional; it can appear once or not at all.**
- `<!ELEMENT addressbook (address+)>`
  An `<addressbook>` element contains one or more `<address>` elements. You can have as many `<address>` elements as you need, but there has to be at least one. **The plus sign indicates that an item must appear at least once, but can appear any number of times.**
- `<!ELEMENT private-addresses (address*)>`
  A `<private-addresses>` element contains zero or more `<address>` elements. **The asterisk indicates that an item can appear any number of times, including zero.**
- `<!ELEMENT name (title?, first-name, (middle-initial | middle-name)?, last-name)>`
  A `<name>` element contains an optional `<title>` element, followed by a `<first-name>` element, possibly followed by either a `<middle-initial>` or a `<middle-name>` element, followed by a `<last-name>` element. In other words, both `<middle-initial>` and `<middle-name>` are optional, and you can have only one of the two. **Vertical bars indicate a list of choices; you can choose only one item from the list.** Also notice that this example uses parentheses to group certain elements, and it uses a question mark against the group.
- `<!ELEMENT name ((title?, first-name, last-name) | (surname, mothers-name, given-name))>`
  The `<name>` element can contain one of two sequences: An optional `<title>`, followed by a `<first-name>` and a `<last-name>`; or a `<surname>`, a `<mothers-name>`, and a `<given-name>`.

A word about flexibility

Before going on, a quick note about designing XML document types for flexibility. Consider the sample name and address document type; I clearly wrote it with U.S. postal addresses in mind.

If you want a DTD or schema that defines rules for other types of addresses, you would have to add a lot more complexity to it. Requiring a `<state>` element might make sense in Australia, but it wouldn't in the UK. A Canadian address might be handled by the sample DTD in Document Type Definitions, but adding a `<province>` element is a better idea. Finally, be aware that in many parts of the world, concepts like title, first name, and last name don't make sense.

The bottom line: If you're going to define the structure of an XML document, you should put as much forethought into your DTD or schema as you would if you were designing a database schema or a data structure in an application. The more future requirements you can foresee, the easier and cheaper it will be for you to implement them later.

Defining attributes

This introductory tutorial doesn't go into great detail about how DTDs work, but there's one more basic topic to cover here: defining attributes. You can define attributes for the elements that will appear in your XML document. Using a DTD, you can also:

- Define which attributes are required
- Define default values for attributes
- List all of the valid values for a given attribute

Suppose that you want to change the DTD to make `state` an attribute of the `<city>` element. Here's how to do that:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city state CDATA #REQUIRED>
```

This defines the `<city>` element as before, but the revised example also uses an `ATTLIST` declaration to list the attributes of the element. The name `city` inside the attribute list tells the parser that these attributes are defined for the `<city>` element. The name `state` is the name of the attribute, and the keywords `CDATA` and `#REQUIRED` tell the parser that the `state` attribute contains text and is required (if it's optional, `CDATA #IMPLIED` will do the trick).

To define multiple attributes for an element, write the `ATTLIST` like this:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city state CDATA #REQUIRED
               postal-code CDATA #REQUIRED>
```

This example defines both `state` and `postal-code` as attributes of the `<city>` element.

Finally, DTDs allow you to define default values for attributes and enumerate all of the valid values for an attribute:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city state CDATA (AZ|CA|NV|OR|UT|WA) "CA">
```

The example here indicates that it only supports addresses from the states of Arizona (AZ), California (CA), Nevada (NV), Oregon (OR), Utah (UT), and Washington (WA), and that the default

state is California. Thus, you can do a very limited form of data validation. While this is a useful function, it's a small subset of what you can do with XML schemas (see XML schemas).

XML schemas

With XML schemas, you have more power to define what valid XML documents look like. They have several advantages over DTDs:

- **XML schemas use XML syntax.** In other words, an XML schema is an XML document. That means you can process a schema just like any other document. For example, you can write an XSLT style sheet that converts an XML schema into a Web form complete with automatically generated JavaScript code that validates the data as you enter it.
- **XML schemas support datatypes.** While DTDs do support datatypes, it's clear those datatypes were developed from a publishing perspective. XML schemas support all of the original datatypes from DTDs (things like IDs and ID references). They also support integers, floating point numbers, dates, times, strings, URLs, and other datatypes useful for data processing and validation.
- **XML schemas are extensible.** In addition to the datatypes defined in the XML schema specification, you can also create your own, and you can derive new datatypes based on other datatypes.
- **XML schemas have more expressive power.** For example, with XML schemas you can define that the value of any `<state>` attribute can't be longer than 2 characters, or that the value of any `<postal-code>` element must match the regular expression `[0-9]{5}(-[0-9]{4})?`. You can't do either of those things with DTDs.

A sample XML schema

Here's an XML schema that matches the original name and address DTD. It adds two constraints: The value of the `<state>` element must be exactly two characters long and the value of the `<postal-code>` element must match the regular expression `[0-9]{5}(-[0-9]{4})?`. Although the schema is much longer than the DTD, it expresses more clearly what a valid document looks like. Here's the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="name"/>
        <xsd:element ref="street"/>
        <xsd:element ref="city"/>
        <xsd:element ref="state"/>
        <xsd:element ref="postal-code"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="name">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="title" minOccurs="0"/>
        <xsd:element ref="first-Name"/>
        <xsd:element ref="last-Name"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
  </xsd:element>

  <xsd:element name="title"      type="xsd:string"/>
  <xsd:element name="first-Name" type="xsd:string"/>
  <xsd:element name="last-Name"  type="xsd:string"/>
  <xsd:element name="street"     type="xsd:string"/>
  <xsd:element name="city"       type="xsd:string"/>

  <xsd:element name="state">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:length value="2"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>

  <xsd:element name="postal-code">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Defining elements in schemas

The XML schema in A sample XML schema defined a number of XML elements with the `<xsd:element>` element. The first two elements defined, `<address>` and `<name>`, are composed of other elements. The `<xsd:sequence>` element defines the sequence of elements that are contained in each. Here's an example:

```
<xsd:element name="address">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="name"/>
      <xsd:element ref="street"/>
      <xsd:element ref="city"/>
      <xsd:element ref="state"/>
      <xsd:element ref="postal-code"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

As in the DTD version, the XML schema example defines that an `<address>` contains a `<name>`, a `<street>`, a `<city>`, a `<state>`, and a `<postal-code>` element, in that order. Notice that the schema actually defines a new datatype with the `<xsd:complexType>` element.

Most of the elements contain text; defining them is simple. You merely declare the new element, and give it a datatype of `xsd:string`:

```
<xsd:element name="title"      type="xsd:string"/>
<xsd:element name="first-Name" type="xsd:string"/>
<xsd:element name="last-Name"  type="xsd:string"/>
<xsd:element name="street"     type="xsd:string"/>
<xsd:element name="city"       type="xsd:string"/>
```

Defining element content in schemas

The sample schema defines constraints for the content of two elements: The content of a `<state>` element must be two characters long, and the content of a `<postal-code>` element must match the regular expression `[0-9]{5}(-[0-9]{4})?`. Here's how to do that:

```
<xsd:element name="state">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="postal-code">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

For the `<state>` and `<postal-code>` elements, the schema defines new data types with restrictions. The first case uses the `<xsd:length>` element, and the second uses the `<xsd:pattern>` element to define a regular expression that this element must match.

This summary only scratches the surface of what XML schemas can do; there are entire books written on the subject. For the purpose of this introduction, suffice to say that XML schemas are a very powerful and flexible way to describe what a valid XML document looks like.

XML programming interfaces

Overview: XML programming interfaces

This section takes a look at a variety of programming interfaces for XML. These interfaces give developers a consistent interface for working with XML documents. There are many APIs available; this section looks at four of the most popular and generally useful ones: the Document Object Model (DOM), the Simple API for XML (SAX), JDOM, and the Java API for XML Parsing (JAXP). (You can find much more information about these APIs through the many links in Resources.)

The Document Object Model

The Document Object Model, commonly called the DOM, defines a set of interfaces to the parsed version of an XML document. The parser reads in the entire document and builds an in-memory tree, so your code can then use the DOM interfaces to manipulate the tree. You can move through the tree to see what the original document contained, you can delete sections of the tree, you can rearrange the tree, add new branches, and so on.

The DOM was created by the W3C, and is an Official Recommendation of the consortium.

DOM issues

The DOM provides a rich set of functions that you can use to interpret and manipulate an XML document, but those functions come at a price. As the original DOM for XML documents was being developed, a number of people on the XML-DEV mailing list voiced concerns about it:

- The DOM builds an in-memory tree of an entire document. If the document is very large, this requires a significant amount of memory.
- The DOM creates objects that represent everything in the original document, including elements, text, attributes, and whitespace. If you only care about a small portion of the original document, it's extremely wasteful to create all those objects that will never be used.
- A DOM parser has to read the entire document before your code gets control. For very large documents, this could cause a significant delay.

These are merely issues raised by the design of the Document Object Model; despite these concerns, **the DOM API is a very useful way to parse XML documents.**

The Simple API for XML

To get around the DOM issues, the XML-DEV participants (led by David Megginson) created the SAX interface. SAX has several characteristics that address the concerns about the DOM:

- A SAX parser sends events to your code. The parser tells you when it finds the start of an element, the end of an element, text, the start or end of the document, and so on. You decide which events are important to you, and you decide what kind of data structures you want to create to hold the data from those events. If you don't explicitly save the data from an event, it's discarded.
- A SAX parser doesn't create any objects at all, it simply delivers events to your application. If you want to create objects based on those events, that's up to you.
- A SAX parser starts delivering events to you as soon as the parse begins. Your code will get an event when the parser finds the start of the document, when it finds the start of an element, when it finds text, and so on. Your application starts generating results right away; you don't have to wait until the entire document has been parsed. Even better, if you're only looking for certain things in the document, your code can throw an exception once it's found what it's looking for. The exception stops the SAX parser, and your code can do whatever it needs to do with the data it has found.

Having said all of these things, both SAX and DOM have their place. The remainder of this section discusses why you might want to use one interface or the other.

SAX issues

To be fair, SAX parsers also have issues that can cause concern:

- SAX events are stateless. When the SAX parser finds text in an XML document, it sends an event to your code. That event simply gives you the text that was found; it does not tell you what element contains that text. If you want to know that, you have to write the state management code yourself.

- SAX events are not permanent. If your application needs a data structure that models the XML document, you have to write that code yourself. If you need to access data from a SAX event, and you didn't store that data in your code, you have to parse the document again.
- SAX is not controlled by a centrally managed organization. Although **this has not caused a problem to date**, some developers would feel more comfortable if SAX were controlled by an organization such as the W3C.

JDOM

Frustrated by the difficulty in doing certain tasks with the DOM and SAX models, Jason Hunter and Brett McLaughlin created the JDOM package. JDOM is a Java technology-based, open source project that attempts to follow the 80/20 rule: Deliver what 80% of users need with 20% of the functions in DOM and SAX. JDOM works with SAX and DOM parsers, so it's implemented as a relatively small set of Java classes.

The main feature of JDOM is that it greatly reduces the amount of code you have to write. Although this introductory tutorial doesn't discuss programming topics in depth, JDOM applications are typically one-third as long as DOM applications, and about half as long as SAX applications. (DOM purists, of course, suggest that learning and using the DOM is good discipline that will pay off in the long run.) JDOM doesn't do everything, but for most of the parsing you want to do, it's probably just the thing.

The Java API for XML Parsing

Although DOM, SAX, and JDOM provide standard interfaces for most common tasks, there are still several things they don't address. For example, the process of creating a `DOMParser` object in a Java program differs from one DOM parser to the next. To fix this problem, Sun has released JAXP, the Java API for XML Parsing. This API provides common interfaces for processing XML documents using DOM, SAX, and XSLT.

JAXP provides interfaces such as the `DocumentBuilderFactory` and the `DocumentBuilder` that provide a standard interface to different parsers. There are also methods that allow you to control whether the underlying parser is namespace-aware and whether it uses a DTD or schema to validate the XML document.

Which interface is right for you?

To determine which programming interface is right for you, you need to understand the design points of all of the interfaces, and you need to understand what your application needs to do with the XML documents you're going to process. Consider these questions to help you find the right approach.

- **Will your application be written in Java?** JAXP works with DOM, SAX, and JDOM; if you're writing your code in Java, you should use JAXP to isolate your code from the implementation details of various parsers.
- **How will your application be deployed?** If your application is going to be deployed as a Java applet, and you want to minimize the amount of downloaded code, keep in mind that

SAX parsers are smaller than DOM parsers. Also be aware that using JDOM requires a small amount of code in addition to the SAX or DOM parser.

- **Once you parse the XML document, will you need to access that data many times?** If you need to go back to the parsed version of the XML file, DOM is probably the right choice. When a SAX event is fired, it's up to you (the developer) to save it somehow if you need it later. If you need to access an event you didn't save, you have to parse the file again. DOM saves all of the data automatically.
- **Do you need just a few things from the XML source?** If you only need a few things out of the XML source, SAX is probably the right choice. SAX doesn't create objects for everything in the source document; you can decide what's important. With SAX, you can look at each event to see if it's relevant to your needs, then process it appropriately. Even better, once you've found what you're looking for, your code can throw an exception to stop the SAX parser altogether.
- **Are you working on a machine with very little memory?** If so, SAX is your best choice, despite all the other factors that you might consider.

Be aware that XML APIs exist for other languages; the Perl and Python communities in particular have very good XML tools.

XML standards

Overview: XML standards

A variety of standards exist in the XML universe. In addition to the base XML standard, other standards define schemas, style sheets, links, Web services, security, and other important items. This section covers the most popular standards for XML, and points you to references to find other standards.

The XML specification

This spec, located at w3.org/TR/REC-xml, defines the basic rules for XML documents. All of the XML document rules discussed earlier in this tutorial are defined here.

In addition to the basic XML standard, the Namespaces spec is another important part of XML. You can find the namespaces standard at the W3C as well: w3.org/TR/REC-xml-names/.

XML Schema

The XML Schema language is defined in three parts:

- **A primer**, located at w3.org/TR/xmlschema-0, that gives an introduction to XML schema documents and what they're designed to do;
- A standard for **document structures**, located at w3.org/TR/xmlschema-1, that illustrates how to define the structure of XML documents;
- A standard for **data types**, located at w3.org/TR/xmlschema-2, that defines some common data types and rules for creating new ones.

This tutorial discussed schemas briefly in Defining document content ; if you want the complete details on all the things you can do with XML schemas, the primer is the best place to start.

XSL, XSLT, and XPath

The Extensible Stylesheet Language, XSL, defines a set of elements (called formatting objects) that describe how data should be formatted. For clarity, this standard is often referred to as XSL-FO to distinguish it from XSLT. Although it's primarily designed for generating high-quality printable documents, you can also use formatting objects to generate audio files from XML. The XSL-FO standard is at w3.org/TR/xsl/.

The Extensible Stylesheet Language for Transformations, XSLT, is an XML vocabulary that describes how to convert an XML document into something else. The standard is at w3.org/TR/xslt (no closing slash).

XPath, the XML Path Language, is a syntax that describes locations in XML documents. You use XPath in XSLT style sheets to describe which portion of an XML document you want to transform. XPath is used in other XML standards as well, which is why it is a separate standard from XSLT. XPath is defined at w3.org/TR/xpath (no closing slash).

DOM

The Document Object Model defines how an XML document is converted to an in-memory tree structure. The DOM is defined in a number of specifications at the W3C:

- **The Core DOM** defines the DOM itself, the tree structure, and the kinds of nodes and exceptions your code will find as it moves through the tree. The complete spec is at w3.org/TR/DOM-Level-2-Core/.
- **Events** defines the events that can happen to the tree, and how those events are processed. This specification is an attempt to reconcile the differences in the object models supported by Netscape and Internet Explorer since Version 4 of those browsers. This spec is at w3.org/TR/DOM-Level-2-Events/.
- **Style** defines how XSLT style sheets and CSS style sheets can be accessed by a program. This spec is at w3.org/TR/DOM-Level-2-Style/.
- **Traversals and Ranges** define interfaces that allow programs to traverse the tree or define a range of nodes in the tree. You can find the complete spec at w3.org/TR/DOM-Level-2-Traversal-Range/.
- **Views** defines an `AbstractView` interface for the document itself. See w3.org/TR/DOM-Level-2-Views/ for more information.

SAX, JDOM, and JAXP

The Simple API for XML defines the events and interfaces used to interact with a SAX-compliant XML parser. You can find the complete SAX specification at www.saxproject.org.

The JDOM project was created by Jason Hunter and Brett McLaughlin and lives at jdom.org/. At the JDOM site, you can find code, sample programs, and other tools to help you get started. (For *developerWorks* articles on JDOM, see Resources.

One significant point about SAX and JDOM is that both of them came from the XML developer community, not a standards body. Their wide acceptance is a tribute to the active participation of XML developers worldwide.

You can find out everything there is to know about JAXP at java.sun.com/xml/jaxp/.

Linking and referencing

There are two standards for linking and referencing in the XML world: XLink and XPointer:

- **XLink,** the XML Linking Language, defines a variety of ways to link different resources together. You can do normal point-to-point links (as with the HTML `<a>` element) or extended links, which can include multipoint links, links through third parties, and rules that define what it means to follow a given link. The XLink standard is at w3.org/TR/xlink/.
- **XPointer,** the XML Pointer Language, uses XPath as a way to reference other resources. It also includes some extensions to XPath. You can find the spec at www.w3.org/TR/xptr/.

Security

There are two significant standards that address the security of XML documents. One is the **XML Digital Signature** standard (w3.org/TR/xmldsig-core/), which defines an XML document structure for digital signatures. You can create an XML digital signature for any kind of data, whether it's an XML document, an HTML file, plain text, binary data, and so on. You can use the digital signature to verify that a particular file wasn't modified after it was signed. If the data you're signing is an XML document, you can embed the XML document in the signature file itself, which makes processing the data and the signature very simple.

The other standard addresses encrypting XML documents. While it's great that XML documents can be written so that a human can read and understand them, this could mean trouble if a document fell into the wrong hands. The **XML Encryption** standard (w3.org/TR/xmlenc-core/) defines how parts of an XML document can be encrypted.

Using these standards together, you can use XML documents with confidence. I can digitally sign an important XML document, generating a signature that includes the XML document itself. I can then encrypt the document (using my private key and your public key) and send it to you. When you receive it, you can decrypt the document with your private key and my public key; that lets you know that I'm the one who sent the document. (If need be, you can also prove that I sent the document.) Once you've decrypted the document, you can use the digital signature to make sure the document has not been modified in any way.

Web services

Web services are an important new kind of application. A Web service is a piece of code that can be discovered, described, and accessed using XML. There is a great deal of activity in this space, but the three main XML standards for Web services are:

- **SOAP:** Originally the Simple Object Access Protocol, SOAP defines an XML document format that describes how to invoke a method of a remote piece of code. My application creates

an XML document that describes the method I want to invoke, passing it any necessary parameters, and then it sends that XML document across a network to that piece of code. The code receives the XML document, interprets it, invokes the method I requested, then sends back an XML document that describes the results. Version 1.1 of the SOAP spec is at w3.org/TR/SOAP/. Visit w3.org/TR/ to see all of the W3C's SOAP-related activities.

- **WSDL:** The Web Services Description Language is an XML vocabulary that describes a Web service. It's possible to write a piece of code that takes a WSDL document and invokes a Web service it's never seen before. The information in the WSDL file defines the name of the Web service, the names of its methods, the arguments to those methods, and other details. You can find the latest WSDL spec at w3.org/TR/wsdl (no closing slash).
- **UDDI:** The Universal Description, Discovery, and Integration protocol defines a SOAP interface to a registry of Web services. If you have a piece of code that you'd like to deploy as a Web service, the UDDI spec defines how to add the description of your service to the registry. If you're looking for a piece of code that provides a certain function, the UDDI spec defines how to query the registry to find what you want. The source of all things UDDI is uddi.org.

Other standards

A number of other XML standards exist that I don't go into here. In addition to widely-applicable standards like Scalable Vector Graphics (www.w3.org/TR/SVG/) and SMIL, the Synchronized Multimedia Integration Language (www.w3.org/TR/smil20/), there are many industry-specific standards. For example, the HR-XML Consortium has defined a number of XML standards for Human Resources; you can find those standards at hr-xml.org.

Finally, for a good source of XML standards, visit Cover Pages for information on many XML schemas and other resources. This site features standards for a wide variety of industries.

Case studies

Real-world examples

By this point, I hope you're convinced that XML has tremendous potential to revolutionize the way eBusiness works. While potential is great, what really counts are actual results in the marketplace. This section describes three case studies in which organizations have used XML to streamline their business processes and improve their results.

All of the case studies discussed here come from IBM's jStart program. The jStart team exists to help customers use new technologies to solve problems. When a customer agrees to a jStart engagement, the customer receives IBM consulting and development services at a discount, with the understanding that the resulting project will be used as a case study. If you'd like to see more case studies, including case studies involving web services and other new technologies, visit the jStart web page.

Be aware that the jStart team is no longer doing engagements for XML projects; the team's current focus is Web services engagements. Web services use XML in a specialized way, typically through the SOAP, WSDL, and UDDI standards mentioned earlier in Web services.

Province of Manitoba
Figure 2. Province of Manitoba

The government of the Province of Manitoba created the Personal Property Registry to provide property owners with state-of-the-art Internet services around the clock. The main benefits of the application were faster and more convenient access to property data, fewer manual steps in the property management process, and fewer calls to the government's call center. In other words, giving customers better service while saving the government money and reducing the government's workload.

Application design

The application was designed as an *n* -tiered application, with the interface separated from the back-end logic. The data for each transaction needed to be transformed a number of different ways, depending on how it needed to be rendered on a device, presented to an application, or formatted for the back-end processing system. In other words, the application was a perfect opportunity to use XML.

As with any application, the user interface to the application was extremely important. To simplify the first implementation, the necessary XML data was transformed into HTML. This gave users a browser interface to the application. The registry was built with VisualAge for Java, specifically the Visual Servlet Builder component. It also uses Enterprise Java Beans (EJBs), including Session beans and Entity beans.

Generating multiple user interfaces with XML

In addition to the HTML interface, a Java client interface and a B2B electronic interface were planned as well. For all of these interfaces, the structured XML data is transformed into the appropriate structures and documents. The initial rollout of the service allowed one business partner, Canadian Securities Registration Systems, to submit XML transaction data using the Secure Sockets Layer. The XML transaction data was transformed into the appropriate format for the back-end transactions.

The end result is that the Province of Manitoba was able to create a flexible new application and their end users could access the property registry more easily and quickly. Because the province uses XML as the data format, the government IT team has a great deal of flexibility in designing new interfaces and access methods. Best of all, the back-end systems didn't have to change at all.

First Union banks on XML
Figure 3. First Union banks on XML

First Union National Bank, one of the largest banks in the U.S., is in the process of reengineering many of its applications using Java and XML. Like most large companies, its environment is heterogeneous, with OS/390, AIX, Solaris, HP/9000, and Windows NT servers and Windows NT, Windows 98, Solaris, and AIX clients. Given this environment, First Union chose Java for platform-independent code and XML for platform-independent data.

A messaging-based system

The bank's distributed applications are built on a messaging infrastructure, using IBM's MQSeries to deliver messages to the OS/390 system. The message content is based on a specification called the Common Interface Message (CIM), a First Union proprietary standard. Both the front-end and back-end components of the application are dependent on the message format. Using XML as the data format isolates both sides of the application from future changes and additions to the messaging protocol.

Using XML tools to automate data flows

In developing this XML-based application, the First Union and IBM team created a service that converts the CIM into an XML document. Another part of the application converts the XML request into the appropriate format for the back-end processing systems. Finally, a third service converts COBOL copy books into DTDs. Once the copy book has been converted into a DTD, First Union can use the DTD and the XML4J parser to validate the XML document automatically; the bank can then be sure that the XML document matches the COBOL data structure that OS/390 expects.

Using Java technology and XML has been very successful for First Union. According to Bill Barnett, Manager of the Distributed Object Integration Team at First Union, "The combination of Java and XML really delivered for us. Without a platform-independent environment like Java and the message protocol independence we received from the use of XML, we would not have the confidence that our distributed infrastructure could evolve to meet the demand from our ever-growing customer base."

Hewitt Associates LLC

Figure 4. Hewitt Associates LLC

Hewitt Associates LLC is a global management consulting firm that specifies in human resource solutions. The company has more than 200 corporate clients worldwide; those companies in turn have more than 10 million employees. Hewitt's clients demand timely, accurate delivery of human resources information for those 10 million employees.

Prior to its jStart engagement, Hewitt built custom, proprietary solutions when its clients requested human resources data. Those custom solutions were typically gateways to Hewitt's existing legacy applications; in some cases, the solutions dealt with the actual byte streams. These custom applications were very costly to develop, test, and deploy, leading Hewitt to investigate Web services.

Web services to the rescue!

To solve these problems, Hewitt and the jStart team worked together to build Web services to address the needs of Hewitt's customers. Web services are a new kind of application that uses XML in a number of interesting ways:

- First of all, Web services typically use SOAP, an XML standard for moving XML data from one place to another.
- Secondly, the interfaces provided by a Web service (method names, parameters, data types, etc.) are described with XML.
- Next, a Web service's description can be stored in or retrieved from a UDDI registry; all of the information that goes into or comes out of the registry is formatted as XML.
- Finally, the data that is provided by the Web service is *itself* XML.

Hewitt has delivered two applications that illustrate their ability to deliver data in more flexible ways:

- With the Secure Participant Mailbox, authorized users can request reports containing personalized information on retirement and other employee benefits.
- With the Retirement Access B2B Connection, authorized users can get details of a client's 401(k) financial information.

Both of these applications retrieve data from existing legacy systems, use XML to format the data, and deliver the formatted information across the Web. Because these applications are built on open standards, Hewitt can deliver them quickly. Best of all, the flexibility of these applications helps Hewitt distinguish itself from its competitors.

"We see Web services as the vehicle to provide open, non-proprietary access to our participant business services and data through a ubiquitous data network," said Tim Hilgenberg, Chief Technology Strategist at Hewitt. The end result: Hewitt develops more flexible applications faster and cheaper, clients get better access to their data, and Hewitt's existing legacy applications don't have to change.

Case studies summary

In all of these case studies, companies used XML to create a system-independent data format. The XML documents represent structured data that can be moved from one system or process to another. As front-end and back-end applications change, the XML traveling between them remains constant. Even better, as more front-end and back-end applications are added to the mix, the use of XML insulates the existing applications from any changes. As Web services become more common, XML will also be used to transport the data.

Advice

Let's go!

At this point, I hope you're convinced that XML is the best way to move and manipulate structured data. If you're not using XML already, how do you get started? Here are some suggestions:

- **Decide what data you want to convert to XML.** Typically this is data that needs to be moved from one system to another, or data that has to be transformed into a variety of formats.
- **See if there are any existing XML standards.** If you're looking at very common data, such as purchase orders, medical records, or stock quotes, chances are good that someone out there has already defined XML standards for that data.
- **See if your existing tools support XML.** If you're using a recent version of a database package, a spreadsheet, or some other data management tool, it's likely that your existing tools (or upgrades to them) can use XML as an input or output format.
- **Learn how to build XML-based applications.** You need to understand how your data is stored now, how it needs to be transformed, and how to integrate your XML development efforts with your existing applications. Benoît Marchal's *Working XML* column is a great place to start; you can find a current listing of all his columns at http://www.ibm.com/developerworks/views/xml/libraryview.jsp?search_by=working+xml:.
- **Join the appropriate standards groups.** Consider joining groups like the World-Wide Web Consortium (W3C), as well as industry-specific groups like HR-XML.org. Being a member of these groups will help you keep track of what's happening in the industry, and it gives you the chance to shape the future of XML standards.
- **Avoid proprietary shenanigans.** It's important that you use only standards-based technology in your development efforts; resist the lures of vendors who offer so-called improvements to you. One of XML's advantages is that you have complete control of your data. Once it's held hostage by a proprietary data format, you've given up a tremendous amount of control.
- **Contact the jStart team.** If you think your enterprise could work with the jStart engagement model, contact the team to see what your possibilities are.
- **Stay tuned to *developerWorks*.** Our XML zone has thousands of pages of content that deal with various XML topics, including DTD and schema development, XML programming, and creating XSLT style sheets.

Resources

- **The dW XML zone** is your one-stop shop for XML resources. See www.ibm.com/developerworks/xml for everything you always wanted to know about XML.
- *developerWorks* has "Fill your XML toolbox" articles that describe XML programming tools for a variety of languages:
  - **C/C++:** See Rick Parrish's article at www.ibm.com/developerworks/library/x-ctlbx.html (*developerWorks*, September 2001).
  - **Perl:** See Parand Tony Darugar's article at www.ibm.com/developerworks/library/x-perl-xml-toolkit/index.html (*developerWorks*, June 2001).
- **XML tutorials:** Dozens of tutorials on XML topics are available on *developerWorks*; see http://www.ibm.com/developerworks/views/xml/libraryview.jsp?type_by=Tutorials for the latest list.
- **IBM's jStart team:** The jStart team works at very low cost to help customers build solutions using new technology (XML Web services, for example). In return, those customers agree to let IBM publicize their projects as a case studies. For more information, see ibm.com/software/jstart.
- **XML standards:** Here's an alphabetical list of all the XML standards mentioned in this tutorial.
  - **DOM**, the Document Object Model:
    - **Core Specification:**w3.org/TR/DOM-Level-2-Core/
    - **Events Specification:**w3.org/TR/DOM-Level-2-Events/
    - **Style Specification:**w3.org/TR/DOM-Level-2-Style/
    - **Traversal and Range Specification:**w3.org/TR/DOM-Level-2-Traversal-Range/
    - **Views Specification:**w3.org/TR/DOM-Level-2-Views/
  - **HR-XML.org**, the Human Resources XML Consortium: hr-xml.org
  - **JAXP**, the Java API for XML: java.sun.com/xml/jaxp/
  - **JDOM**, which doesn't stand for anything: jdom.org/
  - **SAX**, the Simple API for XML: saxproject.org/
  - **SMIL**, the Synchronized Multimedia Integration Language: www.w3.org/TR/smil20/
  - **SOAP**, which used to stand for the Simple Object Access Protocol, but now officially doesn't stand for anything: w3.org/TR/SOAP/
  - **SVG**, Scalable Vector Graphics: www.w3.org/TR/SVG/
  - **UDDI**, the Universal Description, Discovery, and Integration Protocol: uddi.org
  - **WSDL**, the Web Services Description Language: w3.org/TR/wsdl (no closing slash)
  - **XLink**, the XML Linking Language: w3.org/TR/xlink/
  - **XML**, the standard that started it all: w3.org/TR/REC-xml
  - **XML Digital Signatures**: w3.org/TR/xmldsig-core/
  - **XML Encryption**: w3.org/TR/xmlenc-core/
  - **XML Namespaces**: w3.org/TR/REC-xml-names/
  - **XML Repository** of DTDs and schemas: xml.org/xml/registry.jsp
  - **XML Schema**:
    - Part 0 - **Primer**: w3.org/TR/xmlschema-0
    - Part 1 - **Document structures**: w3.org/TR/xmlschema-1
    - Part 2 - **Datatypes**: w3.org/TR/xmlschema-2

- **XPath**, the XML Path Language: w3.org/TR/xpath (no closing slash)
- **XPointer**, the XML Pointer Language: www.w3.org/TR/xptr/
- **XSL-FO**, the Extensible Stylesheet Language for Formatting Objects: w3.org/TR/xsl/
- **XSLT**, the Extensible Stylesheet Language: w3.org/TR/xslt (no closing slash)

About the author

Doug Tidwell

Senior Programmer Doug Tidwell is IBM's evangelist for Web Services. He was a speaker at the first XML conference in 1997, and has been working with markup languages for more than a decade. He holds a Bachelors Degree in English from the University of Georgia and a Masters Degree in Computer Science from Vanderbilt University. He can be reached at dtidwell@us.ibm.com. You can also see his Web page at ibm.com/developerWorks/speakers/dtidwell/.