

XML Schema Validation in Xerces-Java 2

Learn the capabilities of XML Schema

Skill Level: Intermediate

[Nicholas Chase \(nicholas@nicholaschase.com\)](mailto:nicholas@nicholaschase.com)

Author

16 Jul 2002

This tutorial is for developers who need to build XML Schema support into Xerces-Java based applications. It discusses the XML Schema capabilities of the Xerces-Java 2.x parser and demonstrates their use in a Java application.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is for developers who need to build XML Schema support into Xerces-Java based applications. It discusses the XML Schema capabilities of the Xerces-Java 2.x parser and demonstrates their use in a Java application.

Familiarity with XML and Java technology is required. The tutorial covers the basics of the W3C XML Schema recommendation, so while it is helpful, no previous XML Schema knowledge is required. For basic XML information, see the [XML programming in Java technology](#) tutorial. An understanding of XML namespaces is also helpful; you can find basic background in the [Understanding DOM](#) tutorial.

What is this tutorial about?

The original XML specification included Document Type Definitions, or DTDs, which

enabled developers to define a grammar, or structure, for their data. Validating parsers, such as IBM's XML4J or its descendant, Xerces, could then validate XML data against this grammar. The W3C XML Schema specification is an attempt to improve on the DTD -- eliminating some of its weaknesses -- and to allow developers to create XML grammars using XML syntax.

Several schema proposals exist, but this tutorial deals with the W3C XML Schema recommendation; this is the specification I mean when this tutorial mentions *XML Schema* (with an uppercase *S*). When I use the lowercase *schema* or *schemas*, I am referring to documents that I've written or that you might write that conform to the W3C XML Schema Recommendation.

Xerces-Java 1.x provided basic support for XML Schemas, but version 2.x offers essentially complete XML Schema support. This tutorial explains how to use the XML Schema-related features and properties of Xerces-Java to validate documents against an XML Schema document.

The tutorial begins with a quick review of the Document Object Model (DOM) and the Simple API for XML (SAX), and explains how to create a parser for each using Xerces 2.2.0.

Tools

Make sure that the following tools are installed and tested before beginning the tutorial:

- **A Java Virtual Machine**, such as the IBM Java machine or Sun's JDK 1.2 or higher must be installed and working on the target machine. You can find links to JVMs for various platforms in [Resources](#). Users of Java 2 version 1.4.x need to make use of the [The Endorsed Standards Override Mechanism](#).
- **The Xerces-Java 2.0 binary files**: Apache provides precompiled files, so you don't need the source files. Download the [binaries](#).
- **A plain text editor** for creating Java technology applications. If you choose to use an IDE such as VisualAge instead, make sure that the Xerces *.jar files are on the classpath, and that they precede any other XML APIs. (WebSphere Studio includes support for Xerces-J as part of the default install.)

The Endorsed Standards Override Mechanism

Many of the DOM- and SAX-related classes included in Xerces-J are also part of

Java 1.4, and under normal circumstances, the versions included with Java will be used instead of those provided with Xerces.

Fortunately, these classes are "endorsed standards," so you can override the Java version using the **Endorsed Standards Override Mechanism** by placing the relevant jar files (in this case, `xercesImpl.jar`) in the appropriate directory. The default location is

```
%JAVA_HOME%\lib\endorsed
```

on Windows, or

```
%JAVA_HOME%/lib/endorsed
```

on a Linux system.

Developers can also choose a new location by setting the `java.endorsed.dirs` system property.

Users of older versions of Java need not be concerned with this issue.

Section 2. Validating parser basics

What is a parser?

XML data, by definition, consists of tags and content, otherwise known as markup and character data. This predictability makes it readable by humans, who can write and read the data without assistance from an application. Take, for example, this XML document of quotations:

```
<?xml version="1.0"?>
<quotations
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound
      they make as they fly by.
    </saying>
    <attribution>Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding
```

```
    of the universe, why it is as it is and why
    it exists at all.
  </saying>
  <attribution>Stephen Hawking</attribution>
  <era>Modern</era>
</quote>
<quote quoteid="U1">
  <saying>
    Some people make things happen, some watch while
    things happen, and some wonder "What happened?"
  </saying>
  <attribution>Unknown</attribution>
</quote>
</quotations>
```

Using this data in an application involves processing it to provide an in-memory structure of parents and children or a sequence of events that represent elements, attributes, and data. The parser performs this processing.

What is validation?

For a parser to process XML data, the data must be *well-formed*. In other words, it must conform to the rules of XML, such as matching start and end tags, quotation marks around attribute values, properly nested tags, and so on.

Such a document, however, is not necessarily valid.

A *valid* document, in the context of XML, is one that has been successfully compared to a predefined grammar. For example, the data in the previous panel might be described using the following rules:

- The document may have a `quotations` root element.
- The `quotations` element must have one or more `quote` elements.
- A `quote` element must have `saying`, `attribution`, and `era` elements, and a `quoteid` attribute.
- The `saying`, `attribution`, and `era` elements contain text.

Note that according to these rules, the document is invalid, even though it's well-formed, because the third quote does not include an `era` element. The process of comparing a document to its predetermined grammar is called *validation*.

Defining a grammar typically involves the use of either a Document Type Definition (DTD) or an XML Schema document.

DTD basics

Document Type Definitions, or DTDs, are part of the World Wide Web Consortium's original XML 1.0 recommendation, and come straight from XML's predecessor, Standard Generalized Markup Language, or SGML. The original way to define a grammar, DTDs do the job but they require developers to learn a new syntax. For example, the rules in the previous panel can be built into an internal DTD subset as:

```
<?xml version="1.0"?>
<!DOCTYPE quotations [
  <!ELEMENT quotations (quote*)>
  <!ELEMENT quote (saying, attribution, era)>
  <!ATTLIST quote quoteid CDATA #REQUIRED>
  <!ELEMENT saying (#PCDATA)>
  <!ELEMENT attribution (#PCDATA)>
  <!ELEMENT era (#PCDATA)>
]>

<quotations>
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as they fly by.
    </saying>
    ...
  </quote>
</quotations>
```

Because DTDs are part of the XML standard, a validating XML parser must be able to validate a document against them. Some parsers, such as Xerces, also support validation against XML schemas.

Schema basics

XML Schemas convey much the same information as DTDs, but they do it by encoding the information in an XML document. For example, the set of sample rules could be turned into a schema document as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="quotations">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="quote" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="saying" type="xs:string"/>
              <xs:element name="attribution" type="xs:string"/>
              <xs:element name="era" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="quoteid" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The schema in this example imposes the same restrictions as the DTD version, but the XML Schema actually allows for much more control. For example, you could write a schema to restrict the values of the `era` element to a fixed set, such as

Modern, Victorian, and Biblical.

Later in this tutorial, I'll examine several options for specifying the location of a schema document, but for now keep in mind that the schema location can be specified on the root element of the XML document, also known as the *instance document*, as shown in the sample XML data in [What is a parser?](#).

Xerces allows developers to control schema validation from within a Java technology application by setting features on the parser object. The next section discusses the creation and use of parsers from within an application.

Section 3. Xerces parsing basics

The Document Object Model (DOM)

Perhaps the most common API for working with XML is the W3C's Document Object Model, or DOM. DOM actually predates XML: Its first version (also known as Level 0) was implemented in HTML browsers in an attempt to improve scripting capabilities.

Now well on its way to Level 3, DOM represents XML data in a parent-child node structure. The information set is represented as a `Document` object in which each information item can be accessed through methods.

You can create a DOM Document from an XML file within Xerces by either of the following ways:

- Create a `DocumentBuilder` using `DocumentBuilderFactory` as shown below
- Work with the Xerces `DOMParser` object directly (see [Using a DOM parser](#))

The first, and most commonly used, is to use a `DocumentBuilderFactory` to create a `DocumentBuilder`, which then parses the file to create the `Document`, as seen in this example:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import javax.xml.parsers.ParserConfigurationException;
```

```

public class DocBuildDemo {

    public static void main (String args[]){
        DocumentBuilderFactory dbf = null;
        DocumentBuilder db = null;
        Document doc = null;
        try {
            dbf = DocumentBuilderFactory.newInstance();
            db = dbf.newDocumentBuilder();
        } catch (ParserConfigurationException pce) {
            System.out.print("Could not replace parser: ");
            System.out.println(pce.getMessage());
        }
        try {

            doc = db.parse("quotations.xml");

        } catch (java.io.IOException ie){
            System.out.println("Could not read file.");
        } catch (SAXException e) {
            System.out.print("Could not create Document: ");
            System.out.println(e.getMessage());
        }

        DOMDisplay.display(doc);
    }
}

```

In this case, the `DocumentBuilderFactory` object creates the `DocumentBuilder`. The `DocumentBuilder` acts as the parser and returns a `Document` object. In this example, a static method in the `DOMDisplay` class, included in [resource files](#), sends the elements and attributes of the `Document` object to the screen.

This example shows an implementation of JAXP classes within Xerces. While in many cases this is the most convenient way to retrieve the data from an XML file or stream, functionality here is limited to turning validation on and off. The `DOMParser` and `SAXParser` objects, discussed next, offer more control over individual features.

Using a DOM parser

Another way to create a DOM Document object is to work with the Xerces `DOMParser` object directly, as shown here:

```

import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import java.io.IOException;

public class DOMParserDemo {

    public static void main (String args[]){
        DOMParser parser = new DOMParser();
        Document doc = null;
        try {
            parser.parse("quotations.xml");
            doc = parser.getDocument();
        } catch (IOException ie){

```

```

        System.out.println("Could not read file.");
    } catch (SAXException e) {
        System.out.print("Could not create Document: ");
        System.out.println(e.getMessage());
    }

    DOMDisplay.display(doc);
}
}

```

Several differences exist between using the `DocumentBuilder` and using the `DOMParser`. For example, rather than using a factory to create the `DOMParser`, you directly instantiate it. Also, whereas the `DocumentBuilder`'s `parse` method directly returns the `Document` object, the `DOMParser`'s `parse` method does the work internally and doesn't return the document until it is specifically requested through the `getDocument()` method.

After retrieving the `Document`, use `DOMDisplay` to show the information, as shown in [Working with a DOM Document](#).

Working with a DOM Document

Working with a DOM document involves parent and child nodes and their values. For a complete discussion of the DOM, see the tutorial "Understanding DOM" in [Resources](#); however, it's also useful to take a quick look at the `DOMDisplay` class, adapted from there.

```

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.NamedNodeMap;

public class DOMDisplay {

    public static void display (Node start)
    {
        if (start.getNodeType() == start.ELEMENT_NODE)
        {
            System.out.print("<"+start.getNodeName());
            NamedNodeMap startAttr = start.getAttributes();
            for (int i = 0; i < startAttr.getLength(); i++) {
                Node attr = startAttr.item(i);
                System.out.print(" "+attr.getNodeName()+
                    "="+attr.getNodeValue()+"\");
            }
            System.out.print(">");
        } else if (start.getNodeType() == start.TEXT_NODE) {
            System.out.print(start.getNodeValue());
        }

        for (Node child = start.getFirstChild();
            child != null;
            child = child.getNextSibling())
        {

```

```
        display(child);
    }
    if (start.getNodeType() == start.ELEMENT_NODE)
    {
        System.out.print("</"+start.getNodeName()+">");
    }
}
}
```

The application contains a single method, `display()`, which is called recursively to output all of the elements and attributes in the document.

In this case, on the first pass, the method takes the document object itself as an input node. (Remember that all information in a DOM Document is expressed as a particular type of `Node` object.)

First, check the node type. Because the document node is not an element, the value of the node is output. For a document node, this value is null so there is no effect on the output.

Next, recursively call the method for each of the node's children. In the case of this document node, there is just one child -- the root element.

Because the root element is an element, you need to create a start tag for it by outputting an open bracket (`<`), the name of the node, and any attributes. Retrieve the element's attribute nodes as a list using the `getAttributes()` method. Once the start-tag is complete, recursively call the `display()` method for all of the element's children. Text content comes through in its own node, and because it's not an element it is displayed directly. As each element's children are completed, close the element.

Working with a DOM Document (continued)

For simplicity's sake, the method described in [Working with a DOM Document](#) does not take into consideration the formatting concerns of other node types, such as comments and processing instructions, but will accurately display all elements and attributes:

Figure 1. Image of three coded quotations with no special programming

```

<quotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespac
eSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution>Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </saying>
    <attribution>Stephen Hawking</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="U1">
    <saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </saying>
    <attribution>Unknown</attribution>
  </quote>
</quotations>

```

Note that the *pretty printing* effect is achieved not through any special programming, but because the spacing and indents in the original file come through as text nodes, which are also reproduced.

The Simple API for XML (SAX)

The Simple API for XML, or SAX, offers an entirely different way of looking at XML data. Rather than navigating through parent-child nodes, SAX looks at XML data as a stream of events. For example, the XML data shown here:

```

<?xml version="1.0"?>
<quote quoteid="M3">
  <saying>Look before you leap.</saying>
  <attribution>Unknown</attribution>
</quote>

```

consists of 13 events:

```

Start document
Start element (quote)
Characters (whitespace)
Start element (saying)
Characters (Look before you leap.)
End element (saying)
Characters (whitespace)
Start element (attribution)
Characters (unknown)
End element (attribution)
Characters (whitespace)
End element (quote)
End document

```

A SAX parser generates these events (among others), which are sent to a *handler*

and processed accordingly.

Using a SAXParser

Using the `SAXParser` class is fairly straightforward, as demonstrated here:

```
import org.apache.xerces.parsers.SAXParser;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserDemo extends DefaultHandler {

    public static void main(String argv[]) {
        SAXParser parser = new SAXParser();
        parser.setContentHandler(new SAXParserContentHandler());
        parser.setErrorHandler(new SAXParserErrorHandler());

        try {
            parser.parse("quotations.xml");
        } catch (IOException e) {
            System.out.println("Can't read the file.");
        } catch (SAXException e) {
            System.out.println("Parsing error.");
        }
    }
}
```

First, instantiate the `SAXParser` object directly to create the parser. The parser itself will generate events, but if the events are to have any effect, a `ContentHandler` object must process them. This `ContentHandler` can be an instance of the main class (`SAXParserDemo`). To keep the application simple, this example shows both the `ContentHandler` and the `ErrorHandler` broken out into separate classes.

Using an ErrorHandler

The `ErrorHandler` is the simpler of the two. It simply listens for one of three events -- `warning`, `error`, and `fatalError` -- and executes the proper method as the events occur, as shown below:

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

public class SAXParserErrorHandler extends DefaultHandler {

    public SAXParserErrorHandler() {}

    public void warning(SAXParseException ex) throws SAXException {
        System.out.println("WARNING: " + ex.getMessage());
    }

    public void error(SAXParseException ex) throws SAXException {
        System.out.println("ERROR: " + ex.getMessage());
    }
}
```

```
public void fatalError(SAXParseException ex) throws SAXException {
    System.out.println("FATAL ERROR: " + ex.getMessage());
}
}
```

The ContentHandler is a bit more complex, but only because it has more to do.

Using a ContentHandler

The ContentHandler class created for this example echoes the document back to the screen, just as the DOMDisplay class did earlier:

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserContentHandler extends DefaultHandler {

    public SAXParserContentHandler() {}

    public void startDocument() throws SAXException {
        System.out.println("<?xml version=\"1.0\"?>");
    }

    public void startElement(String nsURI, String localName,
                             String qName, Attributes attrs)
                             throws SAXException {
        System.out.print("<"+qName);
        for (int i=0; i < attrs.getLength(); i++) {
            System.out.print(" "+attrs.getQName(i));
            System.out.print("=\""+attrs.getValue(i)+"\"");
        }
        System.out.print(">");
    }

    public void characters(char[] ch, int start, int length)
                             throws SAXException {
        System.out.print(new String(ch, start, length));
    }

    public void ignorableWhitespace(char[] ch, int start, int length)
                             throws SAXException {
        System.out.print(new String(ch, start, length));
    }

    public void endElement(String nsURI, String localName, String qName)
                             throws SAXException {
        System.out.print("</"+qName+">");
    }
}
```

Note that this class has no main() or static methods. It can't be called on its own. Instead, it becomes useful only when another class, such as SAXParserDemo, instantiates it. When it is set as the ContentHandler for the parser, the parser calls each method as each event occurs, feeding it the appropriate information. The output is similar to the DOM output, but has two significant differences:

Figure 2. Image of three coded quotations with XML declaration

```

<?xml version="1.0"?>
<quotations xsi:noNamespaceSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution>Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </saying>
    <attribution>Stephen Hawking</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="U1">
    <saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </saying>
    <attribution>Unknown</attribution>
  </quote>
</quotations>

```

Notice that the XML declaration is present at the top of the output. It's important to realize that this information was not inherent in the parsing of the document and was instead inserted when the `startDocument()` event was generated.

The second factor to keep in mind is that the `SAXParser` is, by default, namespace-aware, so it doesn't represent the namespace declaration on the `quotations` element as an attribute, as the `DOMParser` did. Attributes and elements may also have namespace information included, allowing a developer to choose between the qualified name and the local name. If there is no namespace alias, as in the example above, these two values are the same.

XML Information Set

Because parsing affects the data present within XML independent of validation, it pays to have an understanding of the terminology.

Even experienced XML developers, in many cases, have not heard of the W3C's XML Information Set Recommendation. It's not an API, or a program, or an extension to what can be done with XML. In fact, it doesn't do anything at all, acting instead as a common terminology to which all XML-related standards and recommendations can refer.

For example, what do you call the collection of XML information shown in the preceding examples? Is it a tree? A document? A node? A file? A stream? As demonstrated, it depends on the environment in which you're working. A DOM

parser might refer to it as a `Document` or `Node` object, while a SAX parser would see it as a stream of information. Some schema proposals or other applications liken it to a tree.

The XML Information Set Recommendation seeks to eliminate these terminology problems by creating a neutral set of terms, defining the overall XML document as an *information set*. An information set is a collection of *information items*, each of which has a set of properties determined by the type of information item. For example, an element information item has a property for the namespace name and the local name, among others, as well as properties that represent lists of information items, such as children and attributes.

The process of parsing often involves the addition of information, such as default values and namespace information, to the data. This is known as information set augmentation, and it can happen in the presence of a DTD or schema, even if validation is explicitly disabled, as seen in [The pipeline architecture](#).

The pipeline architecture

Xerces parsing can be likened to a pipeline, where each validator comes through in turn, acting on the data as it is processed. The standard parser configuration includes, as noted in the XML 1.0 recommendation, a DTD validator as part of the pipeline. This means that whether or not the parser reports validation errors, it still validates the document.

Let's look at that statement for a moment: *Whether or not the parser reports validation errors, it still validates the document*. In fact, when a developer speaks of "turning on validation," he or she is actually talking about turning on validation error reporting. But why?

This goes back to the infoset augmentation discussed in [XML Information Set](#). The DTD (and later, the schema) can contain valuable information about the structure of the document, including default and fixed values for attributes. For example, a DTD for the quotations example might read:

```
<!ELEMENT quotations (quote*)>
<!ELEMENT quote (saying, attribution,
                 era
                 )>
<!ATTLIST quote quoteid CDATA #REQUIRED>
<!ELEMENT saying (#PCDATA)>
<!ELEMENT attribution (#PCDATA)>
<!ATTLIST attribution confirmed CDATA "no">
<!ELEMENT era (#PCDATA)>
```

This attribute list definition creates a default value for the confirmed attribute, so if it's not included on any `attribution` element, the infoset is augmented to contain this

information when the DTD is present, as in:

```
<?xml version="1.0"?>
    <!DOCTYPE quotations SYSTEM "quotations.dtd">
    <quotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="quotations.xsd">
        <quote quoteid="A42">
        ...
```

Note that the content model for the `quote` element requires an `era` element, so the document, or information set, is not valid according to this DTD. When the document is parsed, however, no errors are reported, even though the default attribute is added, because validation is not turned on:

Figure 3. Image of three coded quotations without validation turned on

```
<quotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespac
eSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution confirmed="no">Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </saying>
    <attribution confirmed="no">Stephen Hawking</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="U1">
    <saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </saying>
    <attribution confirmed="no">Unknown</attribution>
  </quote>
</quotations>
```

Making use of XML schemas, either for validation or simply for infotset augmentation, requires the manipulation of the pipeline (see [Xerces validation features](#)).

Section 4. Xerces validation features

Validation and Xerces features

Xerces uses a *features* architecture, which allows developers to turn certain capabilities on and off using the `setFeature()` method and a specific URI that

represents the feature. For example, turning on validation involves setting the validation feature, `http://xml.org/sax/features/validation`, to `true`:

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import org.xml.sax.SAXNotRecognizedException;
import org.xml.sax.SAXNotSupportedException;
import java.io.IOException;

public class DOMParserDemo {

    public static void main (String args[]){
        DOMParser parser = new DOMParser();
        Document doc = null;

        try {
            parser.setFeature("http://xml.org/sax/features/validation",true);
        } catch (SAXNotRecognizedException e) {
            System.out.print("Unrecognized feature: ");
            System.out.println("http://xml.org/sax/features/validation");
        } catch (SAXNotSupportedException e) {
            System.out.print("Unrecognized feature: ");
            System.out.println("http://xml.org/sax/features/validation");
        }

        try {
            parser.parse("quotations.xml");
            doc = parser.getDocument();
        } catch (IOException ie){
            System.out.println("Could not read file.");
        } catch (SAXException e) {
            System.out.print("Could not create Document: ");
            System.out.println(e.getMessage());
        }

        DOMDisplay.display(doc);
    }
}
```

Note that even though this is the `DOMParser` application, the potential errors are related to SAX. The designation of `DOMParser` versus `SAXParser` refers to what the parser ultimately produces, and not how it goes about doing it. Both Xerces parsers use SAX to parse the document, so the potential exceptions are SAX exceptions.

With validation turned on and no other features explicitly set, the application now reports DTD-related errors, in addition to adding the defaulted attribute values:

Figure 4. Image of three coded quotations with validation error

```

[Error] quotations.xml:4:68: Attribute "xmlns:xsi" must be declared for element
type "quotations".
[Error] quotations.xml:4:68: Attribute "xsi:noNamespaceSchemaLocation" must be d
eclared for element type "quotations".
[Error] quotations.xml:27:13: The content of element type "quote" is incomplete,
it must match "<saying,attribution,era>".
<quotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespac
eSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution confirmed="no">Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </saying>
    <attribution confirmed="no">Stephen Hawking</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="U1">
    <saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </saying>
    <attribution confirmed="no">Unknown</attribution>
  </quote>
</quotations>

```

XML Schema validation

[Validation and Xerces features](#) demonstrated a parser with validation turned on and a DTD grammar specified. To validate against an XML Schema grammar, remove the DOCTYPE from the XML file and turn on the XML Schema feature:

```

...

public class DOMParserDemo {

    private static void setFeature(String
        feature, boolean setting){

        try {
            parser.setFeature(feature, setting);
        } catch (SAXNotRecognizedException e) {
            System.out.print("Unrecognized feature: ");
            System.out.println(feature);
        } catch (SAXNotSupportedException e) {
            System.out.print("Unrecognized feature: ");
            System.out.println(feature);
        }
    }

    public static DOMParser parser = null;

    public static void main (String args[]){

        parser = new DOMParser();
        Document doc = null;

        setFeature("http://xml.org/sax/features/validation", true);
    }
}

```

```

setFeature("http://apache.org/xml/features/validation/schema",true);

try {

    parser.parse("quotations.xml");
    doc = parser.getDocument();

    } catch (IOException ie){
        System.out.println("Could not read file.");
    } catch (SAXException e) {
        System.out.print("Could not create Document: ");
        System.out.println(e.getMessage());
    }

    DOMDisplay.display(doc);
}

```

Note that in order to keep the application readable, the `setFeature()` method handles this function and all of the exception handling.

In this case, because there is no DTD attached, the defaulted `confirmed` attribute (which is not specified in the schema) is not added to the document. On the other hand, because both the validation and schema features are turned on, the schema validator reports its errors:

Figure 5. Image of three coded quotations with schema validation errors

```

[Error] quotations.xml:26:13: cvc-complex-type.2.4.b: The content of element 'qu
ote' is not complete. It must match '<<"":saying>,<"":attribution>,<"":era>>'.

&ltquotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespac
eSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution>Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="H3">
    <saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </saying>
    <attribution>Stephen Hawking</attribution>
    <era>Modern</era>
  </quote>
  <quote quoteid="U1">
    <saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </saying>
    <attribution>Unknown</attribution>
  </quote>
</quotations>

```

Combining DTDs and schemas

Even when there is no DTD, the DTD validator is still in the pipeline, so how do the two interact? Ultimately, it depends on what grammars are present in the document. Assuming that the validation, namespaces, and schema features are turned on, the

parser behaves as follows:

- For each grammar present, that validator reports errors. This means that if the document specifies only a schema, only the schema validator reports errors. If both a DTD and schema are present, both validators report errors.*
- If neither grammar is present, the last validator in the pipeline (in most cases the schema validator) reports errors.

***Note:** The initial release of Xerces 2.0.1 does not behave reliably when validating against both a DTD and schema. Subsequent releases should fix this and other problems.

Conditional validation

As the application stands right now, it will always report validation errors. If no grammar is present, every element and attribute will be signaled as an error because it has not been declared. One option to handle this problem is to make sure that a grammar is always present and provide a switch that the user can enter on the command line. However, it might be easier to have the application validate the document (and report errors) only if a grammar is present.

Xerces provides the dynamic validation feature for this purpose:

```
...
public class DOMParserDemo {
...
    public static void main (String args[]){
        parser = new DOMParser();
        Document doc = null;

        setFeature("http://xml.org/sax/features/validation", true);
        setFeature("http://apache.org/xml/features/validation/schema", true);
        setFeature("http://apache.org/xml/features/validation/dynamic", true);

        try {

            parser.parse("quotations.xml");
            doc = parser.getDocument();

        } catch (IOException ie){
            System.out.println("Could not read file.");
        } catch (SAXException e) {
            System.out.print("Could not create Document: ");
            System.out.println(e.getMessage());
        }

        DOMDisplay.display(doc);
    }
}
```

In this configuration, both the DTD and schema validators are in the pipeline, but

validation errors are reported only when a grammar is present to validate against.

Other XML Schema-related features in Xerces-J

Xerces-J has a number of other features for developers working with XML Schema. This tutorial does not cover them in detail, but here is a description of the other features. You can learn more about them from the sites listed in [Resources](#) at the end of the tutorial.

- **`http://xml.org/sax/features/validation`** determines whether the parser reports validation errors. The default is `false`.
- **`http://xml.org/sax/features/namespace`** determines whether the parser is namespace-aware. The default is `true`.
- **`http://apache.org/xml/features/validation/schema`** determines whether a schema validator is added to the pipeline. The default is `false`.
- **`http://apache.org/xml/features/validation/dynamic`** provides a means for allowing the application to determine whether to perform validation based on the presence of a grammar. The default is `false`.
- **`http://apache.org/xml/features/validation/schema/element-default`** determines whether the information set is augmented with element defaults (similar to the attribute defaults of DTDs). The default is `true`.
- **`http://apache.org/xml/features/validation/schema-full-checking`** determines whether the schema document itself is checked for certain deep-down problems. The default is `false`.
- **`http://apache.org/xml/features/validation/schema/normalized-value`** determines whether the parser returns the original XML data (`false`) or the normalized data, where line feeds and white space are converted (`true`). The default is `true`.
- **`http://apache.org/xml/features/validation/schema/augment-psvi`** determines whether the post-schema-validation-infoset is augmented with information in the schema. The default is `true`, but if you don't need it, you can set it to `false` to improve performance.

Note that these features are represented as URIs not because there is necessarily any content at that location, but because they provide a way for vendors to specify *their* features uniquely.

Section 5. Creating and using XML Schemas

Preparing the document

Now that the application is built and ready, it's time to start creating the schema and instance documents. The following examples chronicle the building of structures based on the quotation data introduced earlier in the tutorial.

Because the XML data that conforms to the schema is an instance of the structure, the XML document containing it is known as the *instance document*. This document should also contain a reference to the schema document. For example:

```
<?xml version="1.0"?>
<quotations
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="quotations.xsd">
  <quote quoteid="A42">
    <saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </saying>
    <attribution>Douglas Adams</attribution>
    <era>Modern</era>
  </quote>
  ...
</quotations>
```

This example shows one of the simplest forms of schema reference. To signal the parser that information refers back to a schema, create a namespace for the schema instance. The URI `http://www.w3.org/2001/XMLSchema-instance` is known as the schema instance namespace, and that namespace must be used for this purpose. Remember, it's the actual namespace that's important, and not the alias, so while `xsi` is customary, you may use the alias of your choice. Because the `noNamespaceSchemaLocation` is part of that namespace, the parser knows that it refers to the schema processing.

This example uses `noNamespaceSchemaLocation` because none of the data in the instance document itself (except for the schema reference) actually belongs to a namespace. If this were not the case, the schema reference would be as it is described in [Referencing a target namespace](#).

Simple types

Creating a schema document involves creating a simple XML document that defines

the structures and data types used in the instance document. These types fall into two categories: simple types and complex types. Simple types are data types that consist of character data only. Consider the beginnings of this schema document:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="saying" type="xs:string"/>
  <xs:simpleType name="possibleEras">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Biblical"/>
      <xs:enumeration value="Victorian"/>
      <xs:enumeration value="Modern"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

This example shows the schema root element, with the XML Schema namespace definition. The document contains two types of information:

- elements and attributes that are part of the actual schema
- elements and attributes defined by the schema

Namespaces allow the processor to tell the difference. Any references to information prefixed by `xs:` belongs to the XML Schema namespace, and any references without a prefix belong to the namespace of information defined by the schema.

A schema document can document the structure directly, as it does in [Schema basics](#), or it can create types and then reference them to create a structure.

In this example, the first definition refers to an element, `saying`. The `saying` element can contain only content of the built-in type `string`, as evidenced by the `xs:` prefix. Because the element has no attributes defined and no element content, it fits the definition of a simple type.

Also shown here is a named `simpleType`, `possibleEras`. Because it is simply a value, it is a `simpleType`, but it is also an example of a derived type. Using the built-in `string` type as a base, it restricts its value to one of the three enumerated values provided.

Complex types

If an element contains element content or carries attributes, it is no longer a `simpleType`, and becomes a `complexType`, as shown in this example, which documents the `quote` element type. This type will later be referenced by the definition of the `quote` element itself.

```

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="saying" type="xs:string"/>
  <xs:simpleType name="possibleEras">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Biblical"/>
      <xs:enumeration value="Victorian"/>
      <xs:enumeration value="Modern"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="quoteType">
    <xs:sequence>
      <xs:element name="saying" type="xs:string"/>
      <xs:element name="attribution" type="xs:string"/>
      <xs:element name="era" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="quoteid" type="xs:string"/>
  </xs:complexType>

</xs:schema>

```

Notice the overall structure of a `complexType`. First the definition lists the child elements, and then any attributes are defined. Order matters; the attributes must come last.

This example shows a type that requires the `saying`, `attribution`, and `era` elements to appear in the order specified, as evidenced by the fact that their definitions are contained within a `sequence` element. You can also create such definitions with a `choice` element, which allows the document to use one of the contained elements, or `all`, allowing the element to contain all of them in any order.

Referencing named types

Once the schema has declared and named building blocks, it can reference them within other definitions. For example, the simple and complex types (created in [Simple types](#) and [Complex types](#), respectively) can be combined to fill out the rest of the definition:

```

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="saying" type="xs:string"/>

  <xs:simpleType name="possibleEras">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Biblical"/>
      <xs:enumeration value="Victorian"/>
      <xs:enumeration value="Modern"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="quoteType">
    <xs:sequence>
      <xs:element ref="saying"/>

```

```
        <xs:element name="attribution" type="xs:string"/>
        <xs:element name="era" type="possibleEras"/>
    </xs:sequence>
    <xs:attribute name="quoteid" type="xs:string"/>
</xs:complexType>

<xs:element name="quotations">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="quote" type="quoteType"
                minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

First, rather than directly creating the `saying` element, as it does with the `attribution` and `era` elements, the example references the definition that already exists. In the real world, all three elements probably would have been created first, and then referenced.

Referencing named types (continued)

Next, the `era` element is said to have a type of `possibleEras`. This restricts the content of any `era` element to the three values that were enumerated as part of that type.

Finally, the schema defines the root element, `quotations`. Because it contains elements, it is a `complexType`, and even though it contains only a single type of element, the `sequence` element (or one of its alternatives) must be used. The `quote` element itself is defined directly within the overall definition, but the type is referenced as the `quoteType` created earlier. In this way, the structure is carried forward into this element. The `minOccurs` and `maxOccurs` attributes define how many times an element may appear. If they are not present, the element must appear exactly once. Because `maxOccurs` is set to `unbounded`, it may appear any number of times, but a developer could just as easily choose a finite number of occurrences.

The `minOccurs` attribute is also useful for defining optional elements by setting its value to 0, as seen in the example.

Using a target namespace

The previous examples showed an instance document with no namespace, but this is not always the case. In some situations, the elements and attributes being defined belong to a specific namespace. This namespace becomes the target namespace for the schema.

```

<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/quote.html"
  xmlns:qu="http://www.example.com/quote.html"
  attributeFormDefault="qualified"
  elementFormDefault="qualified">

  <element name="saying" type="string"/>

  <simpleType name="possibleEras">
    <restriction base="string">
      <enumeration value="Biblical"/>
      <enumeration value="Victorian"/>
      <enumeration value="Modern"/>
    </restriction>
  </simpleType>

  <complexType name="quoteType">
    <sequence>
      <element ref="qu:saying"/>
      <element name="attribution" type="string"/>
      <element name="era" type="qu:possibleEras"/>
    </sequence>
    <attribute name="quoteid" type="string"/>
  </complexType>

  <element name="quotations">
    <complexType>
      <sequence>
        <element name="quote" type="qu:quoteType"
          minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>

```

Notice the namespace declarations. First, the Schema namespace is now the default namespace, so I removed all of the `xs:` prefixes. Next, the `targetNamespace` attribute defines the namespace for the eventual data, and the following namespace declaration defines an alias (`qu:`) by which it can be referred. Finally, the `attributeFormDefault` and `elementFormDefault` attributes define whether the elements and attributes in the instance document should be qualified or not.

Note that even if the instance document makes use of the default namespace, the elements will be qualified, but the attributes will not. Attributes never inherit the default namespace and must be explicitly qualified.

Once I made those declarations, I had to change the aliases on the schema elements to convert the previous example to one with a target namespace. The elements and attributes drawn from the XML Schema now bear no alias, as they are part of the default namespace. The elements, attributes, and types the example schema defines are now part of the target namespace, also aliased as `qu:`. Therefore, when elements (such as `saying`) or types (such as `possibleEras` or `quoteType`) are referenced, they must be prefixed.

Referencing a target namespace

Adding the data to the target namespace requires two changes to the instance document:

```
<?xml version="1.0"?>
<qu:quotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/quote.html quotations.xsd"
  xmlns:qu="http://www.example.com/quote.html">
  <qu:quote qu:quoteid="A42">
    <qu:saying>
      I love deadlines. I like the whooshing sound they make as
      they fly by.
    </qu:saying>
    <qu:attribution>Douglas Adams</qu:attribution>
    <qu:era>Modern</qu:era>
  </qu:quote>
  <qu:quote qu:quoteid="H3">
    <qu:saying>
      My goal is simple. It is complete understanding of the
      universe, why it is as it is and why it exists at all.
    </qu:saying>
    <qu:attribution>Stephen Hawking</qu:attribution>
    <qu:era>Modern</qu:era>
  </qu:quote>
  <qu:quote qu:quoteid="U1">
    <qu:saying>
      Some people make things happen, some watch while things
      happen, and some wonder "What happened?"
    </qu:saying>
    <qu:attribution>Unknown</qu:attribution>
    <qu:era>Victorian</qu:era>
  </qu:quote>
</qu:quotations>
```

First, add the data itself to the target namespace. Second, to let the parser know where to find the schema, use the `schemaLocation` attribute. This attribute takes the namespace in question and the location of the schema document, separated by a space. Note that the namespace for the data must match the target namespace specified on the schema document.

The `schemaLocation` attribute makes it possible to combine data from several namespaces by including multiple namespaces and locations, as in:

```
schemaLocation="http://www.example.com/other.html other.xsd
  http://www.example.com/quote.html quotations.xsd"
```

Choosing the schema programmatically

The XML Schema recommendation actually refers to the `noNamespaceSchemaLocation` and `schemaLocation` attributes as *hints* for the processor, which may legitimately ignore them. One situation where this would be desirable would be if the application itself specified the schema location, either

based on business logic or on user input.

You specify this information programmatically much the same as you would set a feature programmatically, but this time you set a property:

```

...
public class DOMParserDemo {
...
    public static void main (String args[]){

        parser = new DOMParser();
        Document doc = null;

        setFeature("http://xml.org/sax/features/validation", true);
        setFeature("http://apache.org/xml/features/validation/schema",true);

        try {
            parser.setProperty(
                "http://apache.org/xml/properties/schema/external-schemaLocation",
                "http://www.example.com/quote.html quotations.xsd");
        } catch (SAXNotRecognizedException e) {
            System.out.print("Unrecognized property: ");
            System.out.println(
                "http://apache.org/xml/properties/schema/external-schemaLocation");
        } catch (SAXNotSupportedException e) {
            System.out.print("Unrecognized property: ");
            System.out.println(
                "http://apache.org/xml/properties/schema/external-schemaLocation");
        }

        try {

            parser.parse("quotations.xml");
            doc = parser.getDocument();

        } catch (IOException ie){
            System.out.println("Could not read file.");
        } catch (SAXException e) {
            System.out.print("Could not create Document: ");
            System.out.println(e.getMessage());
        }

        DOMDisplay.display(doc);
    }
}

```

For data with no namespace, use the

`http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation` property instead.

Section 6. Minor XML Schema limitations in Xerces-J

Known limitations

The Xerces developers have identified a handful of known issues that probably will not interfere with any developer's ability to use the package, but should be kept in mind.

- The following values may not be validated correctly by the parser if they are larger than 2147483647: `length`, `minLength`, and `maxLength` facets, and `year` and `second` values in date/time datatypes.
- Large values of `maxOccurs` may cause a `StackOverflowError`. The current workaround is to change the `maxOccurs` value to `unbounded`.
- Certain decimal values that should be invalid may not cause an error because of changes in the specification of a `Decimal` in Java 1.3.

Some issues are related not to errors or limitations, but to ambiguity within the XML Schema recommendation. These include a lack of definition of a unit for length of a value of type `QName` and questions over the definition of `keys` and `keyrefs`.

In the case of a `QName`, the length is assumed to be the number of characters in the namespace URI plus one character for the colon (:) plus the number of characters in the local name.

In the case of `keys` and `keyrefs`, the Xerces implementation requires a `keyref` to refer to a `key` or `unique` within the scope of the element to which the `keyref` is attached. This requirement is stricter than the XML Schema Part 1: Structures Recommendation would imply, and should be kept in mind when building the schema document

Bugs and other limitations

It has been some time since the initial release of Xerces-Java 2, and it appears that as of version 2.2.0, most of the bugs have been taken care of. For an up-to-date look at existing bugs and their status, check out Bugzilla at <http://nagoya.apache.org/bugzilla/query.cgi>.

Section 7. Summary

Summary

XML data increases its usefulness if its structure can be predicted. It is for this

reason that the XML 1.0 recommendation provided for Document Type Definitions against which a validating parser could compare the data. These DTDs provided for additional information, such as defaulted attributes, but also required the learning of a new syntax, and were limited in the amount of control provided to programmers. For these reasons, the XML Schema specification was developed. Schemas allow the definition of XML structures using XML itself.

The Xerces-Java 2 XML parser is a fully-conforming XML Schema processor. It enables developers to create schemas according to the W3C's XML Schema Recommendation and validate instance documents accordingly. Xerces provides a good deal of programmatic control over the process through the use of features and properties that can be set within the application.

Resources

Learn

- For a thorough introduction to important XML concepts, see the "[XML programming in Java technology](#)" tutorial (*developerWorks*, September 1999).
- For a good understanding of the underlying recommendations Xerces uses, see the "[Understanding DOM](#)" (*developerWorks*, September 2001) and "[Understanding SAX](#)" tutorials (*developerWorks*, August 2001).
- Take the "[Validating XML](#)" tutorial for a look at using Xerces to validate an XML document using the W3C's XML schemas (*developerWorks*, September 2001).
- For an introduction to XML schemas, read "[The basics of using XML Schema to define elements](#)," by Ashvin Radiya and Vibha Dixit (*developerWorks*, August 2000).
- Robert Costello provides an extremely detailed look at using XML schemas with his XML Schema Tutorial, which you can find at [xFront.com](#).
- For a comparison of DTDs and XML Schemas, read Kevin Williams' discussion, "[Why XML Schema beats DTDs hands-down for data](#)" (*developerWorks*, June 2001).
- Visit XML.org's registry of [industry standard schemas](#).
- Read Paul Golick and Richard Mader's article on schemas and naming conventions in "[Real World XML Schema](#)" (*developerWorks*, January 2002).
- For a look at other XML schema proposals, see Robin Cover's [schema resources](#).
- For information on creating schemas using WebSphere Studio Application Developer, see "[XML and WebSphere Studio Application Developer, Part 1: Developing XML Schema](#)" by Christina Lau.
- For a complete guide to using Xerces to build XML applications, see [XML and Java from Scratch](#) , by Nicholas Chase, the author of this tutorial.
- Xerces API documentation is included with the distribution, or at [the Apache project](#).
- Find out how you can become an [IBM Certified Developer in XML and related technologies](#).
- Explore many more XML resources on the [developerWorks XML zone](#).
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Download the [resource files](#) for the `DOMDisplay` class mentioned in [The Document Object Model \(DOM\)](#).
- Download IBM's [Schema Quality Checker](#) from alphaWorks (free download with 90-day license, renewable). Schema Quality Checker evaluates your schema documents for errors.
- Download Xerces-Java from [the Apache project](#).
- Download versions of Xerces for [C++](#) and [Perl](#).
- Download versions of Java for various OSes from <http://www.ibm.com/developerworks/java/jdk/index.html>.
- Download Windows, Linux x86, and Solaris/SPARC x86 versions of Java from <http://java.sun.com/j2se/1.3/>.
- Download a Mac OS X version of Java from <http://devworld.apple.com/java/>.
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content.](#)

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, 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, USA, and is the author of three books on Web development, including *Java and XML From Scratch* (Que) and the upcoming *Primer Plus XML Programming* (Sams). He loves to hear from readers and can be reached at nicholas@nicholaschase.com.