XML programming in Java technology, Part 2

Skill Level: Introductory

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

09 Jul 2004

This advanced tutorial covers more sophisticated topics for manipulating XML documents with Java technology. Author Doug Tidwell shows you how to do tasks such as generate XML data structures, validate XML documents, work with namespaces, and interface XML parsers with non-XML data sources. As you'd expect, all of the examples are based on open standards.

Section 1. Introduction

About this tutorial

In an earlier tutorial ("XML programming in Java technology, Part 1"), I showed you the basics of XML parsing in the Java language. I covered the major APIs (DOM, SAX, and JDOM), and went through a number of examples that demonstrated the basic tasks common to most XML applications. This tutorial will look at more difficult things that weren't covered before, such as:

- Getting and setting parser features
- Working with namespaces
- Validating XML documents

As in the introductory tutorial, the APIs I'll cover are:

- The Document Object Model (DOM), Levels 1, 2, and 3
- The Simple API for XML (SAX), Version 2.0
- JDOM, a simple Java API created by Jason Hunter and Brett McLaughlin

- The Java API for XML Processing (JAXP)

I'll also cover several approaches to validation, including W3C XML Schema, RELAX NG, and Schematron.

About the examples

Most of the examples here will work with the Shakespearean sonnet that appeared in the last tutorial. The structure of this sonnet is:

```
<sonnet>
  <author>
    <lastName>
    <firstName>
    <nationality>
    <yearOfBirth>
    <yearOfDeath>
  </author>
  <lines>
    [14 <line> elements]
  </lines>
</sonnet>
```

In the various sample programs, some versions of this document will have namespaces, and some will use DTDs, W3C XML Schemas, or other schema languages for validation. For the complete examples, see the following files:

- sonnet.xml
- sonnet.dtd (download to view in a text editor)
- sonnetNamespaces.xml
- sonnet.xsd
- sonnetSchema.xml
- sonnet.rng
- sonnetRules.xsl
- sonnetSchematron.xml

As an alternative, download x-java2_code_files.zip to view these files in a text editor.

Setting up your machine

You'll need to set up a few things on your machine before you can run the examples. (I'm assuming that you know how to compile and run a Java program, and that you know how to set your `CLASSPATH` variable.)

1. First, visit the home page of the Xerces XML parser at the Apache XML Project (http://xml.apache.org/xerces2-j/). You can also go directly to the download page (http://xml.apache.org/xerces2-j/download.cgi).

2. Unzip the file that you downloaded from Apache. This creates a directory named `xerces-2_5_0` or something similar, depending on the release level of the parser. The JAR files you need (`xercesImpl.jar` and `xml-apis.jar`) should be in the Xerces root directory.

3. Visit the JDOM project's Web site and download the latest version of JDOM (http://jdom.org/).

4. Unzip the file you unloaded from JDOM. This creates a directory named `jdom-b9` or something similar. The JAR file you need (`jdom.jar`) should be in the `build` directory.

5. Finally, download the zip file of examples for this tutorial, x-java2_code_files.zip , and unzip the file.

6. Add the current directory (`.`), `xercesImpl.jar`, `xml-apis.jar`, and `jdom.jar` to your `CLASSPATH`.

Section 2. Getting and setting parser features

Parser features

As XML has become more sophisticated, parsers have had to become more sophisticated as well. DOM, SAX, and JDOM all define sets of parser features. Some features are required, and some are optional. Each of the three APIs provide similar methods and exceptions for getting and setting parser features.

For example, take a look at SAX. The SAX API itself defines methods to get and set parser features in the `XMLReader` interface. JAXP provides those same methods in the `SAXParserFactory` and `SAXParser` classes. Here's how that code works:

```
public static void checkFeatures()
{
  try
  {
    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setFeature
      ("http://xml.org/sax/features/namespace-prefixes",
       true);
    if (!spf.getFeature
        ("http://xml.org/sax/features/validation"))
      spf.setFeature
        ("http://xml.org/sax/features/validation", true);
    . . .
  }
```

In addition to the `getFeature()` and `setFeature()` methods, JAXP defines methods for working with the commonly-used namespace and validation features. The `SAXParserFactory` class defines the `setNamespaceAware()` and `setValidating()` methods to set those features on any `SAXParser` it creates. In addition, both `SAXParserFactory` and `SAXParser` provide the `isNamespaceAware()` and `isValidating()` methods.

Setting SAX parser features

I left out one detail about setting SAX features `getFeature()` and `setFeature()` methods: handling exceptions. Whenever you're setting SAX parser features, you should always catch two exceptions: `SAXNotSupportedException` and `SAXNotRecognizedException`.

```
catch (SAXNotSupportedException snse)
{
  System.out.println
    ("The feature you requested is not supported.");
}
catch (SAXNotRecognizedException snre)
{
  System.out.println
    ("The feature you requested is not recognized.");
}
```

`SAXNotSupportedException` means that the parser recognizes the feature you've requested, but doesn't support it, while `SAXNotRecognizedException` means that the parser has never heard of the feature you're asking for.

SAX parser features

The SAX 2.0 standard requires that parsers recognize these two features:

`http://xml.org/sax/features/namespaces`
: The parser recognizes namespaces: When this property is `true`, namespace URIs and unqualified local names are available for all elements and attributes. Any SAX 2.0-compliant parser must support the default value of `true` for this property.

`http://xml.org/sax/features/namespace-prefixes`
: The parser provides support for resolving namespace prefixes. When this

property is `true`, namespace prefixes are available for elements and attributes, including `xmlns:` attributes. Any SAX 2.0-compliant parser must support the default value of `false` for this property.

One more point about these two required features: A parser is required to provide `get` methods for them, but does not have to provide `set` methods.

To help you avoid `SAXNotRecognizedException` s, here is a list of commonly-supported features. These are not defined in the SAX 2.0 standard (that standard only defines the two required features). A SAX parser doesn't have to support or recognize any of these features, and any given parser is free to add its own features to this list.

`http://xml.org/sax/features/external-general-entities`
Determines whether the parser processes external general entities. This feature doesn't have a default value, although if validation is turned on, this feature will be `true`.

`http://xml.org/sax/features/external-parameter-entities`
Determines whether the parser processes external parameter entities. This feature doesn't have a default value, although if validation is turned on, this feature will be `true`.

`http://xml.org/sax/features/is-standalone`
This property defines whether the XML declaration contains `standalone="yes"`. It can't be changed, it can only be queried, and it can only be queried after the `startDocument` event.

`http://xml.org/sax/features/lexical-handler/parameter-entities`
The SAX parser's `LexicalHandler` will report the beginning and end of parameter entities.

`http://xml.org/sax/features/resolve-dtd-uris`
A value of `true` indicates that `SYSTEM` IDs used to define declarations will be reported relative to the base URI of the document. `false` indicates that the IDs will be reported as found in the document; programs can use the `Locator.getSystemId()` method to get the document's base URI.

`http://xml.org/sax/features/string-interning`
If this property is `true`, all XML names and namespace URIs are interned using `java.lang.String.intern()`. Using `intern()` makes string comparison much faster than calls to `java.lang.String.equals()`.

`http://xml.org/sax/features/use-attributes2`
`http://xml.org/sax/features/use-locator2`
`http://xml.org/sax/features/use-entity-resolver2`
A value of `true` indicates that the parser uses the updated interfaces `org.xml.sax.ext.Attributes2`, `org.xml.sax.ext.Locator2`, or `org.xml.sax.ext.EntityResolver2`, respectively.

`http://xml.org/sax/features/validation`
Specifies whether the parser is validating the document. If this feature is `true`, `external-general-entities` and `external-parameter-entities` are automatically set to `true`.

`http://xml.org/sax/features/xmlns-uris`
When the `namespace-prefixes` feature is set, this feature controls whether namespace declarations are in namespaces themselves. The Namespaces in XML spec states that namespace declarations are not to be in any namespace, while the DOM Level 1 spec puts namespaces declarations into the namespace `http://www.w3.org/2000/xmlns/`. When this feature is `true`, namespace declarations are in a namespace; when the feature is `false`, they aren't.

You can find a list of all the defined feature URIs at saxproject.org/apidoc/org/xml/sax/package-summary.html.

A brief word about entities

A number of parser features deal with **entities** and how they are handled by the parser. I'm including a brief summary of entities here in case you're not familiar with them. In general, an entity defines a string that is replaced with something else when the XML document is processed. The different types of entities are listed here.

General versus parameter entities: A general entity defines a substitution that is used inside an XML document. A parameter entity, on the other hand, only appears in a DTD, and defines a substitution that can only be used inside a DTD. (More on DTDs later.) A general entity looks like this:

```
<!ENTITY corp "International Business Machines Corporation">
```

Using this entity, the string `&corp;` is replaced with the string `International Business Machines Corporation` wherever it appears.

A parameter entity looks like this:

```
<!DOCTYPE article [
  <!ENTITY % basic "a|b|code|dl|i|ol|ul|#PCDATA">
  <!ELEMENT body (%basic;)*>
```

In this example, the parameter entity `basic` is associated with a particular string. Assuming these are HTML elements, wherever you use the parameter entity `%basic;`, that means an element can contain text (`#PCDATA` stands for "parsed character data") or the elements `<a>`, `<b>`, `<code>`, `<dl>`, `<i>`, `<ol>`, and `<ul>`.

Using this parameter entity throughout the DTD can save a lot of typing.

Internal versus external entities: An internal entity is defined in the XML file; an external entity is defined in a different (external) file. The external file is most likely another XML file, but it could be something else (more on that in a minute). The first line here defines an internal entity, while the second defines an external entity by using the `SYSTEM` keyword and a reference to an external file:

```
<!ENTITY auth "Doug Tidwell">
<!ENTITY BoD  SYSTEM "http://www.ibm.com/board_of_directors.html">
```

Using these examples, the string `&auth;` will be replaced with the text `Doug Tidwell`, while the string `&BoD;` will be replaced with the contents of the file `board_of_directors.html`. The first entity here is an internal general entity, and the second is an external general entity.

Predefined entities: The XML standard defines five entities that are always available. They are the entities for the less-than sign (`&lt;`), the greater-than sign (`&gt;`), the ampersand (`&amp;`), the apostrophe or single-quote character (`&apos;`), and the double-quote character (`&quot;`).

Setting DOM parser features

JAXP's `DocumentBuilderFactory` has a smaller set of features than the `SAXParserFactory` does. The most important methods are:

`setValidating(boolean)`
Sets the factory's validation property.

`isValidating()`
Returns `true` if the factory creates validating parsers, `false` otherwise.

`setNamespaceAware(boolean)`
Sets the factory's namespace-aware property.

`isNamespaceAware()`
Returns `true` if the factory creates namespace-aware parsers, `false` otherwise.

`setIgnoringElementContentWhitespace(boolean)`
Sets the factory's whitespace property. If this is true, the parsers created by the factory won't create nodes for the ignorable whitespace in the document.

`isIgnoringElementContentWhitespace()`
Returns `true` if the factory creates parsers that ignore whitespace, `false` otherwise.

In addition to the above features, the following properties are rarely used:

`setCoalescing(boolean)` and `isCoalescing()`
The factory creates parsers that convert `CDATA` sections into text nodes. (`CDATA` stands for "character data", and refers here to non-parsed text.)

`setExpandEntityReferences(boolean)` and `isExpandEntityReferences()`
The factory creates parsers that expand entity reference nodes.

`setIgnoringComments(boolean)` and `isIgnoringComments()`
The factory creates parsers that ignore comments.

Section 3. An overview of namespaces

Namespaces introduction

When XML was first announced, frustrated HTML developers were thrilled with the prospect of creating their own tags. Finally they could all create tags to describe their data, rather than forcing their data to fit into the narrow structures of HTML.

Once the initial euphoria died down, it became obvious that things weren't quite that simple; sooner or later two or more groups would define the same tag. For example, I run an online bookstore, so I use the `<title>` tag for the title of a book. At your business, you might store the addresses of all your customers in XML, using the `<title>` tag for a customer's courtesy title.

To create an XML book order, it's perfectly reasonable for me to use my tags to describe the books being ordered, and it's perfectly reasonable for you to use your tags to describe the shipping address for the order. The obvious problem: How do we distinguish between these two `<title>` tags? **Namespaces** are the answer.

Conceptually, a namespace works like a Java `package` statement. Two classes can have the same name, as long as they're from two different packages. For example, JDOM and DOM both define a `Document` class. To make it clear *which* class I want to use, I combine the package name and the class name, as in `org.jdom.Document` and `org.w3c.dom.Document`.

A namespace has two parts: A **prefix** and a **unique string**. Here's a fragment of a document that uses namespaces:

```
<bookOrder  xmlns:lit="http://www.literarysociety.org/books"
         xmlns:addr="http://www.usps.com/addresses" >
 . . .
```

```
    <lit:title>My Life in the Bush of Ghosts</lit:title>
    . . .
    <shipTo>
      <addr:title>Ms.</addr:title>
      <addr:firstName>Linda</addr:firstName>
      <addr:lastName>Lovely</addr:lastName>
      . . .
```

This document defines two namespaces. The `lit` prefix is associated with the string `http://www.literarysociety.org/books`, and the `addr` prefix is associated with the string `http://www.usps.com/addresses`. When you use a `<lit:title>` or `<addr:title>` element, it's clear which `<title>` element you're using.

More namespace details

When you define a namespace on a given element, that namespace can be used by that element and every element inside it. In the previous example, I defined all the namespaces I'm using on the root element of the document. I could have defined the `addr` namespace on the `<shipTo>` element:

```
<shipTo xmlns:addr="http://www.usps.com/addresses">
  <addr:title>Ms.</addr:title>
  <addr:firstName>Linda</addr:firstName>
  <addr:lastName>Lovely</addr:lastName>
  . . .
```

If I wanted to type as much as possible, I could redefine the namespace on every single element that uses it:

```
<shipTo>
  <addr:title xmlns:addr="http://www.usps.com/addresses">
    Ms.
  </addr:title>
  <addr:firstName xmlns:addr="http://www.usps.com/addresses">
    Linda
  </addr:firstName>
  <addr:lastName xmlns:addr="http://www.usps.com/addresses">
    Lovely
  </addr:lastName>
  . . .
```

Whenever a namespace prefix is used, the namespace associated with that prefix has to be defined on that element or one of its ancestors. Defining all of the namespaces on the root element simplifies and shortens the document.

A final technique is to use the `xmlns` attribute without defining a prefix at all. This defines the **default namespace** for the current element and any descendant elements that *don't* have a namespace prefix. I could have coded the `<title>` element like this:

```
<title xmlns="http://www.literarysociety.org/books">
  My Life in the Bush of Ghosts
</title>
```

In a moment, I'll show you a namespace-qualified version of the XML sonnet. If you wanted, you could use a default namespace for the `<author>` element:

```
 <author
  xmlns="http://www.literarysociety.org/authors">
  <lastName>Shakespeare</lastName>
  <firstName>William</firstName>
  <nationality>British</nationality>
  <yearOfBirth>1564</yearOfBirth>
  <yearOfDeath>1616</yearOfDeath>
</author>
```

Because none of these elements has a namespace prefix, a namespace-aware parser will report all of these elements as belonging to the `http://www.literarysociety.org/authors` namespace. Outside of the `<author>` element, this default namespace is no longer defined.

Comparing two namespaces

Sometimes you will want to check the value of a namespace. For example, in XSLT stylesheets all of the stylesheet elements have to be from the namespace `http://www.w3.org/1999/XSL/Transform`. Typically, this namespace string is associated with the prefix `xsl`, but that's not required. When you make sure the namespace for a given element is correct, you need to check the namespace *string*, not the namespace prefix. In other words, this XSLT element is correct:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

while this XSLT element is not:

```
<xsl:stylesheet xmlns:xsl="http://i-love-stylesheets.com">
```

In the second example, the namespace prefix is what you'd expect, but the namespace string is wrong. As a final example, this XSLT element is correct:

```
<xqdkera:stylesheet
   xmlns:xqdkera="http://www.w3.org/1999/XSL/Transform">
```

The prefix here isn't the traditional `xsl`, but that's not important. When comparing two namespaces, *the prefix doesn't matter.*

A common misconception about namespaces

The most confusing thing about namespaces is *the unique string is not used as a URL* (yes, it looks like one, but it isn't). It is common practice for groups to use their domain names to make sure the string is unique. You might think (as I first did) that an Internet-connected XML parser would download a DTD or schema from that URL, but it doesn't.

Although the Namespaces in XML standard defines the unique string as a URI (Universal Resource *Identifier* -- see Resources), it's effectively just a string. `xmlns:addr="cranberries"` is legal, as long as `cranberries` is unique among all the namespace definitions in this document. For all the details on valid URIs, see the URI standard (RFC2396) in Resources on page 38.

One final (potentially confusing) point

If you have no trouble understanding that the unique string isn't a URL and that an XML parser will never use that string as a URL, read on. If you have any doubts, or if you're still confused, see Getting namespace information from a parser.

Even though the namespace string isn't a URL, *sometimes* if you point your browser at the string you'll find useful information about a tag set or document type. That's completely optional, and intended for human consumption, not for machines. As an example, the namespace `http://www.w3.org/2003/05/soap-envelope/role/next` is defined in the SOAP 1.2 specification. If you point your browser at http://www.w3.org/2003/05/soap-envelope/role/next, you'll see an HTML page that points you to more information about the SOAP 1.2 spec.

If other people are likely to use the elements you define, it's a good idea to provide this information just in case someone tries to load a namespace URI.

Section 4. Parsing with namespaces

Getting namespace information from a parser

Now that you've seen the basics of namespaces, I'll take a look at DOM, SAX, and JDOM, to see how they report namespace information. All of these examples use the following modified version of the sonnet:

```
<sonnet  pt:type="Shakespearean"
   xmlns:pt="http://www.literarysociety.org/poemtypes" >
 <auth:author
     xmlns:auth="http://www.literarysociety.org/authors">
   <auth:lastName>Shakespeare</auth:lastName>
   <auth:firstName>William</auth:firstName>
   <auth:nationality>British</auth:nationality>
   <auth:yearOfBirth>1564</auth:yearOfBirth>
   <auth:yearOfDeath>1616</auth:yearOfDeath>
  </auth:author>
  <title>Sonnet 130</title>
  <lines>
  . . .
  </lines>
</sonnet>
```

When working with namespaces, you'll need to know several pieces of information for any namespace-qualified element. As an example, consider the `<auth:lastName>` element above:

The local (unqualified) name of the element
: `lastName` -- The name of the element without any namespace prefix

The namespace prefix
: `auth`

The qualified name of the element
: `auth:lastName` -- The name of the element, including any namespace prefix

The namespace URI
: `http://www.literarysociety.org/authors`

I'll show you how each API exposes this information.

DOM and namespaces

Namespace support was added to the Document Object Model in DOM Level 2. Here's a list of your information items, along with the DOM methods that give you that information:

The local (unqualified) name of the element
: `Node.getLocalName()`

The namespace prefix
: `Node.getPrefix()`

The qualified name of the element
: `Node.getNodeName()`

The namespace URI
: `Node.getNamespaceURI()`

Notice that all of these methods are part of the `Node` interface. Be aware that using these methods has a few complications:

- If you invoke `getLocalName()`, `getPrefix()`, or `getNamespaceURI()` against anything other than an `Element` or `Attribute`, the result is `null`.
- If you invoke `getLocalName()`, `getPrefix()`, or `getNamespaceURI()` against an element created with a DOM Level 1 method such as `Document.createElement()`, the result is `null`. (To create a node that can be used with these methods, use the DOM Level 2 method `Document.createElementNS()` instead.)
- For `Element` s and `Attribute` s, `getNodeName()` always returns the name of the element or attribute. If the element or attribute is namespace-qualified, the node name will be `auth:lastName` or something similar; if it is not, the node name will be `lastName`. To determine if a given node name is namespace-qualified, you have to see if `getPrefix()` is non-null or search for a colon in the node name.

Creating a namespace-aware DOM parser

Now take a look at `DomNS`, a Java program from the introductory XML programming in Java tutorial (http://www.ibm.com/developerworks/edu/x-dw-xml-i.html) that's similar in structure to `DomOne`. You'll see two basic differences here: You need to create a namespace-aware DOM parser, and add more code to process any namespace information in the DOM tree. Instead of merely echoing the parsed document back to the console, you print information about any namespace-qualified elements in the DOM tree.

Your first step is to create a namespace-aware parser. For most parsers, namespace-awareness is turned off by default. Because you're using JAXP, you need to set the property of your `DocumentBuilderFactory` before you create your DOM parser:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
 dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
doc = db.parse(uri);
```

```
    if (doc != null)
      printNamespaceInfo(doc.getDocumentElement());
```

Use the JAXP `setNamespaceAware()` method to turn on namespace-awareness for your factory class; from that point, any DOM parser that the factory creates will be namespace-aware. Once you have the parser set up, call the recursive `printNamespaceInfo()` method to find all the namespace-qualified elements and attributes in your document.

Finding namespaces in a DOM tree

The `printNamespaceInfo()` method looks at any elements and attributes in the DOM tree, and prints details of those nodes whenever the `getPrefix()` method returns a non-null string. Here's the bulk of the code:

```
case Node.ELEMENT_NODE:
  {
    if (node.getPrefix() != null)
    {
      System.out.println("\nElement " + node.getNodeName());
      System.out.println("\tLocal name = " +
                         node.getLocalName());
      System.out.println("\tNamespace prefix = " +
                         node.getPrefix());
      System.out.println("\tNamespace URI = " +
                         node.getNamespaceURI());
    }

    if (node.hasAttributes())
    {
      NamedNodeMap attrs = node.getAttributes();
      for (int i = 0; i < attrs.getLength(); i++)
        if ((attrs.item(i).getPrefix()) != null)
                         printNamespaceInfo (attrs.item(i));
    }

    if (node.hasChildNodes())
    {
      NodeList children = node.getChildNodes();
      for (int i = 0; i < children.getLength(); i++)
        printNamespaceInfo (children.item(i));
    }

    break;
  }

case Node.ATTRIBUTE_NODE:
  {
    System.out.println("\nAttribute " +
                       node.getNodeName() + "="
                       + node.getNodeValue());
    System.out.println("\tLocal name = " +
                       node.getLocalName());
    System.out.println("\tNamespace prefix = " +
                       node.getPrefix());
    System.out.println("\tNamespace URI = " +
                       node.getNamespaceURI());
    break;
  }
```

To process element nodes, follow these three steps:

1. If the element has a namespace prefix, print the details for the node.
2. Look at any attributes the element has; if any of them are namespace-qualified, call `printNamespaceInfo()` for them.
3. Invoke `printNamespaceInfo()` for all of the element's children.

To process an attribute node, simply print its details. Notice that `printNamespaceInfo()` is assumed to be invoked only for namespace-qualified attribute nodes.

When you run `DomNS` against the file `sonnetNamespaces.xml`, you get these results:

```
C:\adv-xml-prog>java DomNS sonnetnamespaces.xml

Attribute pt:type=Shakespearean
        Local name = type
        Namespace prefix = pt
        Namespace URI = http://www.literarysociety.org/poemtypes

Attribute xmlns:pt=http://www.literarysociety.org/poemtypes
        Local name = pt
        Namespace prefix = xmlns
        Namespace URI = http://www.w3.org/2000/xmlns/

Element auth:author
        Local name = author
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

Attribute xmlns:auth=http://www.literarysociety.org/authors
        Local name = auth
        Namespace prefix = xmlns
        Namespace URI = http://www.w3.org/2000/xmlns/

Element auth:lastName
        Local name = lastName
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

. . .
```

The output shows details of any namespace-qualified element or attribute in the source document. The partial output here lists the `pt:type` attribute, the `<auth:author>` element, and the `<auth:lastName>` element.

for `xmlns:pt` and `xmlns:auth` map to a namespace of `http://www.w3.org/2000/xmlns/`. This is the default namespace used by DOM parsers for namespace definitions themselves. Section 4 of the original XML namespaces specification stated, "The prefix `xmlns` is used only for namespace bindings and is not itself bound to any namespace name." However, when DOM Level 2 was released, Section 1.1.8 of the DOM Level 2 spec amended this:

Note: In the DOM, all namespace declaration attributes are by definition bound to the namespace URI: `http://www.w3.org/2000/xmlns/`. These are the attributes whose namespace prefix or qualified name is "xmlns." Although, at the time of writing, this is not part of the XML Namespaces specification, it is planned to be incorporated in a future revision.

In other words, any attribute that defines a namespace (`xmlns:abc="..."`) or a default namespace (`xmlns="..."`) will itself be mapped to the namespace `http://www.w3.org/2000/xmlns/`. When you parse the same document with SAX, namespace definitions are not reported as attributes, so no namespace is defined for the definitions themselves.

To see the complete source code, check out DomNS.java.

Namespace-aware DOM methods

To wrap up this discussion of the DOM and namespaces, here's a list all of the namespace-aware DOM methods. A brief description follows the name of each method; check the DOM documentation that came with your parser for all the details on the methods and how they work.

`Document.createAttributeNS(...)`
: Creates an attribute with a given namespace and qualified name.

`Document.createElementNS(...)`
: Creates an element with a given namespace and qualified name.

`Document.getElementsByTagNameNS(...)`
: Returns a `NodeList` with all of the descendant nodes that match a given namespace and local name.

`Element.getAttributeNodeNS(...)`
: Returns an `Attribute` node, given a namespace and a local name for the attribute.

`Element.getAttributeNS(...)`
: Returns the value of an attribute, given a namespace and a local name for the attribute.

`Element.getElementsByTagNameNS(...)`
: Returns a `NodeList` with all of the descendant nodes that match a given namespace and local name.

`Element.hasAttributeNS(...)`
: Returns `true` if this element has an attribute with a given namespace and local name, `false` otherwise.

`Element.removeAttributeNS(...)`
: Given a namespace and a local name, removes the attribute with that namespace and local name from this element.

`Element.setAttributeNodeNS(...)`
: Adds a given namespace-qualified `Attr` object to this element.

`Element.setAttributeNS(...)`
: Given a namespace, the local name of an attribute, and a value, adds to this element a namespace-qualified attribute with the specified value.

`Node.getLocalName()`
: Returns the local (unqualified) name of a given node.

`Node.getNamespaceURI()`
: Returns the namespace string associated with a given node, or `null` if the element or attribute isn't associated with a namespace.

`Node.getNodeName()`
: Returns the name of this node. If the node is namespace-qualified, this method returns the prefix and the element name; otherwise, it returns the element name only.

`Node.getPrefix()`
: Returns the namespace prefix for this node, or `null` if the element or attribute didn't have a prefix in the XML source.

`Node.setPrefix(...)`
: Sets the namespace prefix for this node.

Note: All of the methods defined in the `Node` interface return useful information for `Element` s and `Attribute` s only. Invoking these methods against other node types returns `null`.

SAX and namespaces

As with DOM, the first release of SAX was not namespace-aware. SAX 2 added namespace awareness through a variety of methods; I'll review those here. First, take a look at how you get your four information items:

The local (unqualified) name of the element
: The `localName` parameter (the second parameter) to the `startElement` and `endElement` events.

The namespace prefix

Not directly accessible. The namespace prefix is everything before the colon in the `qName` parameter (the third parameter) to the `startElement` and `endElement` events.

The qualified name of the element
: The `qName` parameter (the third parameter) to the `startElement` and `endElement` events.

The namespace URI
: The `uri` parameter (the first parameter) to the `startElement` and `endElement` events.

As you'd expect, namespace information is provided through various events, particularly `startElement` and `endElement`.

Creating a namespace-aware SAX parser

Now take a look at `SAXNS`, a Java program similar to the earlier `DomNS`. As with `DomNS`, your first step is to create a namespace-aware parser. Set a property of the `SAXParserFactory` object, create a `SAXParser`, and then you're ready to start parsing. Here's how to create the `SAXParser` you need:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
 spf.setNamespaceAware(true);
SAXParser sp = spf.newSAXParser();
sp.parse(uri, this);
```

Use the JAXP `setNamespaceAware()` method to turn on namespace-awareness for your factory class; from that point, any `SAXParser` that the factory creates will be namespace-aware. Once you have the parser set up, your event handlers take it from there.

Finding namespaces in SAX events

For `SaxNS`, you want to echo the details of any namespace-qualified elements or attributes to the console. You can find all the information you need in the `startElement` event. Here's the code for the `startElement` event handler:

```
public void startElement(String namespaceURI, String localName,
                         String qName, Attributes attrs)
{
  if (namespaceURI.length() > 0)
  {
    System.out.println("\nElement " + qName);
    System.out.println("\tLocal name = " + localName);
    if (qName.indexOf(':') > 0)
```

```
        System.out.println("\tNamespace prefix = " +
                          qName.substring(0, qName.indexOf(':')));
      else
        System.out.println("\tNamespace prefix =");
      System.out.println("\tNamespace URI = " + namespaceURI);
    }

    if (attrs != null)
    {
      int len = attrs.getLength();
      for (int i = 0; i < len; i++)
        if (attrs.getURI(i).length() > 0)
        {
          System.out.println("\nAttribute " +
                            attrs.getQName(i) + "=" +
                            attrs.getValue(i));
          System.out.println("\tLocal name = " +
                            attrs.getLocalName(i));
          if (qName.indexOf(':') > 0)
            System.out.println("\tNamespace prefix = " +
                              attrs.getQName(i).
                              substring(0,
                                        attrs.getQName(i).
                                        indexOf(':')));
          else
            System.out.println("\tNamespace prefix = ");
          System.out.println("\tNamespace URI = " +
                            attrs.getURI(i));
        }
    }
  }
```

As you'd expect, this code is very similar to the code for `DomNS`. However, there are a few differences:

- When you're dealing with the properties of an element, those properties are arguments to the event handler. In the DOM version, those properties are accessed through methods.
- As I mentioned earlier, you can't get the namespace prefix of an element or attribute directly. The example code uses the `java.lang.String.indexOf()` and `java.lang.String.substring()` methods to extract the prefix from the qualified name of the element or attribute. (Notice that you have to make sure the qualified name contains a colon; if this element or attribute uses a default namespace, the qualified name won't include a colon.)
- Attributes in SAX work much more like attributes in the DOM world. The attributes of an element are represented as a set of objects, and methods such as `getLocalName()` and `getValue()` let you work with the properties of a given attribute.

When you run `SaxNS` against the file `sonnetNamespaces.xml`, you get these results:

```
C:\adv-xml-prog>java SaxNS sonnetnamespaces.xml

Attribute pt:type=Shakespearean
        Local name = type
        Namespace prefix = pt
        Namespace URI = http://www.literarysociety.org/poemtypes

Element auth:author
        Local name = author
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

Element auth:lastName
        Local name = lastName
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

Element auth:firstName
        Local name = firstName
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

Element auth:nationality
        Local name = nationality
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors
```

For the most part, the output of `SaxNS` is the same as the output from `DomNS`. The only difference is that the `SAXParser` doesn't report the namespace definitions themselves (such as `xmlns:pt="http://www.lit..."`) as attributes. In a minute, I'll show you the SAX events that handle namespace definitions.

For the complete source code, see SaxNS.java.

Namespace-specific SAX events

In the discussion on finding namespaces with a DOM parser (Finding namespaces in a DOM tree on page), I pointed out that all namespace definitions are reported as attributes belonging to the `http://www.w3.org/2000/xmlns/` namespace. You might have noticed that in the output to `SaxNS` namespace definitions didn't show up in the output at all. In other words, the attributes for the `<sonnet>` element included the `pt:type` attribute, but didn't include the definition of the `pt` namespace itself.

them through separate events. The `startPrefixMapping` and `endPrefixMapping` events tell you when a particular namespace is defined, as well as when that namespace goes out of scope and is no longer defined. Next, I'll look at a new application, `SaxNSTwo`, which uses these new events to handle namespaces.

One of the reasons for handling `startPrefixMapping` and `endPrefixMapping` is to keep track of the various prefixes and the namespace URIs that they are mapped to. In `SaxNSTwo`, I show you how to create a private `HashMap` to keep track of namespace events as they come in. Here's the code that implements the

`HashMap` and the event handlers:

```
private HashMap prefixes = new HashMap();
. . .
public void startPrefixMapping(String prefix, String uri)
{
  System.out.println("\nNew namespace:");
  System.out.println("\tNamespace prefix = " + prefix);
  System.out.println("\tNamespace URI = " + uri);

   prefixes.put(uri, prefix);
}

public void endPrefixMapping(String prefix)
{
  System.out.println("\nPrefix " + prefix +
                     " is no longer in scope.");
}
```

The logic here is pretty simple; when you get a `startPrefixMapping` event, you add that prefix and URI combination to the `HashMap`. When you get an `endPrefixMapping` event, you remove the prefix and URI. The last enhancement to make to `SaxNSTwo` is to add a private method to get the prefix mapping for a particular URI:

```
private String getPrefix(String url)
{
  if (prefixes.containsKey(url))
    return prefixes.get(url).toString();
  else
    return "";
}
```

When you're echoing namespace information to the console, you can now use your `getPrefix()` method to retrieve the prefix that's associated with a given URI:

```
if (namespaceURI.length() > 0)
{
  System.out.println("\nElement " + qName);
  System.out.println("\tLocal name = " + localName);
  System.out.println("\tNamespace prefix = " +
                     getPrefix(namespaceURI) );
  System.out.println("\tNamespace URI = " + namespaceURI);
}
```

The complete source code is in SaxNSTwo.java.

A final approach to handling namespaces

The approach used in `SaxNSTwo` was fairly straightforward: When you get a `startPrefixMapping` event, you put an entry into your `HashMap`, then retrieve that value whenever you need it. If a given URI is mapped to two prefixes, and those definitions are nested within each other, `SaxNSTwo` won't get the job done. This case isn't common, but it is legal. `SaxNS Two` doesn't correctly handle the case of a given prefix being mapped to two URIs, but the parser's error checking kicks in before your code does anything wrong.

To handle this problem (assuming you consider it a problem at all), you would need to implement a stack for each URI, pushing a value onto that URI's stack with a `startPrefixMapping` event, and popping a value from that URI's stack with an `endPrefixMapping` event.

If this case matters to you, the `org.xml.sax.helpers` package provides a special class, `NamespaceSupport`, to manage namespaces as they go into and out of scope. I'll look at a final class, `SaxNSThree`, that uses a `NamespaceSupport` object to deal with namespaces.

Here's how to handle namespaces using a `NamespaceSupport` object:

- When you get a `startPrefixMapping` event, call `pushContext()` to store the current set of defined namespaces. After that, call `declarePrefix` to add the newly-defined namespace to the current context.
- When you get an `endPrefixMapping` event, call `popContext` to return to the previous set of namespace definitions.
- Whenever you need to get the prefix associated with a given URI, call `getPrefix()`.

Now take a look at `SaxNSThree`. First of all, here's the declaration of the `NamespaceSupport` object and the two event handlers:

```
private NamespaceSupport ns = new NamespaceSupport();
. . .
public void startPrefixMapping(String prefix, String uri)
{
   ns.pushContext();
  ns.declarePrefix(prefix, uri);
}

public void endPrefixMapping(String prefix)
{
   ns.popContext();
}
```

To illustrate namespace handling, you'll process all of the elements and attributes as before, plus you'll list all of the namespaces defined when you process each element. The output for your sample sonnet looks like this:

```
C:\adv-xml-prog>java SaxNSThree sonnetnamespaces.xml

<sonnet> : 2 prefixes defined - (xml, pt)

Attribute pt:type=Shakespearean
        Local name = type
        Namespace prefix = pt
        Namespace URI = http://www.literarysociety.org/poemtypes

<auth:author> : 3 prefixes defined - (xml, auth, pt)
        Local name = author
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors

<auth:lastName> : 3 prefixes defined - (xml, auth, pt)
        Local name = lastName
        Namespace prefix = auth
        Namespace URI = http://www.literarysociety.org/authors
. . .
<title> : 2 prefixes defined - (xml, pt)

<lines> : 2 prefixes defined - (xml, pt)
. . .
```

Notice that the XML prefix is always defined; it is mapped to the string `http://www.w3.org/XML/1998/namespace`. The `NamespaceSupport` object keeps track of which namespace definitions are currently in scope.

One downside to the implementation of `NamespaceSupport` is that it returns `Enumeration` s for the `getPrefixes()` method. (`NamespaceSupport` uses `Enumeration` s in other places, too.) To get the number of namespaces in scope, you have to write some clumsy code:

```
private int getPrefixCount()
{
  Enumeration e = ns.getPrefixes();
  int count = 0;
  while (e.hasMoreElements())
  {
    count++;
    e.nextElement();
  }

  return count;
}
```

If the `getPrefixes()` method returns a Java collection object of some kind, you can use the `size()` method to get the number of prefixes currently defined.

For the complete source code, see SaxNSThree.java.

Namespace-aware SAX objects

Before I move on to namespace processing with JDOM, here's a list of all the namespace-aware SAX classes and interfaces, along with a brief discussion of each. As always, check the SAX documentation that comes with your XML parser for the final word on the methods and their meanings.

`org.xml.sax.Attributes`
: With the exception of `getLength()`, which returns the number of attributes, all of the methods of this class are namespace-aware.

`org.xml.sax.ContentHandler`
: The `startElement`, `endElement`, `startPrefixMapping`, and `endPrefixMapping` events are all namespace-aware.

`org.xml.sax.helpers.AttributesImpl`
: The following methods are namespace-aware: `addAttribute()`, `getIndex()`, `getLocalName()`, `getQName()`, `getType()`, `getURI()`, `getValue()`, `setAttribute()`, `setLocalName()`, `setQName()`, and `setURI()`.

`org.xml.sax.helpers.DefaultHandler`
: The `startElement`, `endElement`, `startPrefixMapping`, and `endPrefixMapping` events are all namespace-aware. (These events are defined in the `ContentHandler` interface, which is implemented by `DefaultHandler`.)

`org.xml.sax.helpers.NamespaceSupport`
: This class exists to manage namespaces as they go into and out of scope.

`org.xml.sax.helpers.XMLFilterImpl`
: This class implements the `ContentHandler` interface, so it includes the namespace-aware events `startElement`, `endElement`, `startPrefixMapping`, and `endPrefixMapping`.

JDOM and namespaces

To begin this discussion, take a look at the JDOM APIs for getting the four basic pieces of information for an element or attribute:

`Element.getName()`
: The local (unqualified) name of the element

`Element.getNamespacePrefix()`
: The namespace prefix

`Element.getQualifiedName()`
: The qualified name of the element

`Element.getNamespaceURI()`
: The namespace URI

The names of these methods are straightforward. To illustrate how JDOM works with namespaces, I'll show you `JdomNS`, an application that parses an XML file and echoes namespace information back to the console.

When I showed you how to process namespaces with DOM and SAX, I mentioned that you have to specify that you want a namespace-aware parser. With JDOM, namespaces are turned on by default in the underlying SAX parser, encapsulated by the `org.jdom.input.SAXBuilder` object. If you need to control the properties of the `SAXBuilder`, you can use the `setFeature()` or `setProperty()` methods. Be aware that the JDOM needs the SAX parser to be configured a certain way, so the JDOM documentation recommends that you use these methods with caution.

Processing namespace information with JDOM

To build `JdomNS`, parse the XML file and get your JDOM `Document` structure back. As with `DomNS`, you'll have to walk through that structure and find all of the namespace information. You'll use a recursive approach here. Here's how the code begins:

```
SAXBuilder sb = new SAXBuilder();
Document doc = sb.build(new File(argv[0]));
if (doc != null)
  printNamespaceInfo(doc.getRootElement() );
```

Notice that the argument to `printNamespaceInfo()` is a JDOM `Element`. In DOM, you had a single datatype (the `Node`) that was subclassed by every node type. With JDOM, you'll work with `Element` s only.

Speaking of `printNamespaceInfo()`, I'll show you this method next. It has four tasks:

1. If this element is namespace-qualified, echo the namespace information to the console.
2. If this element has any namespace declarations, echo those to the console.
3. If this element has any namespace-qualified attributes, echo the namespace information about those attributes to the console.
4. If this element has any children, invoke `printNamespaceInfo()` against each child.

I'll take these tasks in order as I go through the source code. First of all, to see if the current element is namespace-qualified, check the namespace URI of the current element:

```
if ( el.getNamespaceURI().length() > 0 )
{
  System.out.println("\nElement " + el.getQualifiedName());
  System.out.println("\tLocal name = " + el.getName());
  System.out.println("\tNamespace prefix = " +
                    el.getNamespacePrefix());
  System.out.println("\tNamespace URI = " +
                    el.getNamespaceURI());
}
```

Next, look for any additional namespaces defined on this element. Notice that JDOM handles namespace declarations differently than DOM. In the DOM application, a namespace declaration (`xmlns:tp="http://..."`, for example) was reported as another attribute. With JDOM, the namespace declaration is not considered an attribute, so you have to use the `getAdditionalNamespaces()` method.

A final note: If the current element contains its own namespace definition (such as the `<auth:author xmlns:auth="...">` element in the sample document), that namespace definition is available only through the current element, *not* through `getAdditionalNamespaces()`. Here's the next segment of the code:

```
Iterator nsIter = el. getAdditionalNamespaces().
listIterator();
while (nsIter.hasNext())
{
  Namespace ns =  (Namespace) nsIter.next();
  System.out.println("\nNamespace declaration:");
  System.out.println("\tNamespace prefix = " + ns.getPrefix());
  System.out.println("\tNamespace URI = " + ns.getURI());
}
```

With all of the JDOM methods that return sets of things (`getAttributes()`, `getAdditionalNamespaces()`, `getChildren()`, and so forth), that set of things is returned to you as a `List`, part of the Java Collections API. You should use a `ListIterator` to walk through the `List`. A final note about using collections: Because the `ListIterator` returns `Object` s, you have to cast the various items to `Namespace` s, `Attribute` s, and so forth.

Your next task is to look at all of the current element's attributes and see if any of them are namespace-qualified. That code uses a `ListIterator`, as you'd expect:

```
Iterator attrIter = el.getAttributes().listIterator();
while (attrIter.hasNext())
{
  Attribute attr = (Attribute)attrIter.next();
  if ( attr.getNamespaceURI().length() > 0 )
  {
    System.out.println("\nAttribute " +
                      attr.getQualifiedName() + "=" +
                      attr.getValue());
  . . .
```

The final task is to get the children of this `Element` and call `printNamespaceInfo` to handle each one. Here's how that code looks:

```
  Iterator childIter = el.getChildren().listIterator();
  while (childIter.hasNext())
    printNamespaceInfo((Element)childIter.next());
```

For the complete source code, see JdomNS.java.

Section 5. Validating XML documents

Validation overview

When XML was first introduced, it's validation scheme was the Document Type Definition, or DTD. DTDs came from the SGML world, and are typically concerned with the structure of a document only. They have a limited notion of data typing, but nothing like what you would expect from a modern programming language.

For example, I can use a DTD to define that a `<postcode>` element is required for an `<address>` element, and a validating parser will enforce that. Any XML document that contains an `<address>` without a `<postcode>` will be flagged as not valid. Unfortunately, a parser using a DTD for validation is equally happy with `<postcode>B9C 4F8</postcode>` as it is with `<postcode>Mad dogs and Englishmen</postcode>`. Clearly the XML world needed a more robust validation language.

To fill the void, the World Wide Web Consortium (W3C) created the XML Schema language, an attempt to address the needs of the XML community. XML Schema (I'll call it that from now on) is divided into two parts: data types and document structures. The data types spec defines some basic data types as well as rules for creating new ones, while the document structure spec defines a set of XML tags for specifying what elements a document can contain.

Defining a markup language to describe the contents of an XML document is a daunting task, and as you'd expect, not everyone was/is happy with XML Schema. In this tutorial, I'll show you two other schema languages, RELAX NG and Schematron. Support for these languages isn't as widespread as XML Schema, but each of them has a loyal and active following.

One final point about DTDs: They use a different (and completely incompatible) syntax than XML documents. This is another holdover from the SGML world. Although some people argue that the different syntaxes are a good thing, most users prefer a validation language specifiable as XML. XML Schema, RELAX NG, and Schematron are all XML-based. Among other things, that means you can write an XSLT stylesheet that converts a schema into a human-readable document that explains the rules of the schema.

Defining a document with a DTD

Before I explore validation with various kinds of parsers, take a look at the document definitions you'll use. First of all, the DTD:

```
<!ELEMENT sonnet  (author,title?,lines)>

<!ATTLIST sonnet  type (Shakespearean | Petrarchan)
                  "Shakespearean">

<!ELEMENT author  (lastName,firstName,nationality,
                yearOfBirth?,yearOfDeath?)>
. . .
<!ELEMENT lines (line,line,line,line,
              line,line,line,line,
              line,line,line,line,
              line,line)>

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

The DTD defines all of your elements and attributes. The syntax above defines the `type` attribute of the `<sonnet>` element. It also defines the valid values (`Shakespearean` and `Petrarchan`), as well as the default value (`Shakespearean`). Other syntax notes:

- The question marks next to some of the elements means those elements are optional.
- `#PCDATA` means an element contains only text, not other elements.
- To specify that a `<lines>` element contains 14 `<line>` elements, you have to list all 14 elements. If a `<lines>` element could contain 10, 12, or 14 `<line>` elements, then you would have to list all of the possibilities with 10 `<line>` elements, followed by a vertical bar (`|`), followed by 12 `<line>` elements, followed by a vertical bar, followed by 14 `<line>` elements.

Defining a document with an XML Schema

Next, look at an XML Schema that defines the sonnet document type. The significant parts of the schema are:

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

  <xsd:element name="sonnet">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="author"/>
        <xsd:element ref="title" minOccurs="0"/>
        <xsd:element ref="lines"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:attribute name="type" type="sonnetType"
      default="Shakespearean"/>
  </xsd:element>

  <xsd:simpleType name="sonnetType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Petrarchan"/>
      <xsd:enumeration value="Shakespearean"/>
    </xsd:restriction>
  </xsd:simpleType>
. . .
  <xsd:element name="title" type="xsd:string"/>

  <xsd:element name="lines">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="line" minOccurs="14" maxOccurs="14"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="line" type="xsd:string"/>
```

Several items are defined here. First is the `<sonnet>` element: Its contents are an `<author>` element, a `<title>` element, and a `<lines>` element. Some syntax notes:

- Use `minOccurs="0"` to indicate that the `<title>` element is optional.
- Define a separate datatype for your attribute; that datatype is based on the `xsd:string` datatype, and can contain one of two values, `Shakespearean` or `Petrarchan`.
- On the definition of the attribute itself, the default value of the attribute is also defined.
- To define that a sonnet contains 14 `<line>` s, use `minOccurs="14"` and `maxOccurs="14"`.

Defining a document with RELAX NG

When it comes to defining the structure of a document, XML Schema isn't the only game in town. Another popular method is RELAX NG, a project led by XML guru James Clark. RELAX NG is currently being developed by an OASIS technical committee (see Resources). To quote the committee's Web site:

The purpose of this TC is to create a specification for a schema language that is simple, easy to learn, and uses XML syntax.

True to the committee's design goals, the RELAX NG syntax is very simple. To define an element, you use the `<element>` element. To define that an element or attribute is optional, you use the `<optional>` element. Here's a fragment of the RELAX NG definition for the example sonnet:

```
<grammar>
  . . .
  <start>
    <element name="sonnet">
      <ref name="typeAttribute"/>
      <ref name="author"/>
      <ref name="title"/>
      <ref name="lines"/>
    </element>
  </start>
</grammar>
```

From this simple listing, it looks like a `<sonnet>` element contains four things, defined elsewhere in the RELAX NG file as `typeAttribute`, `author`, `title`, and `lines`. `typeAttribute` is an attribute named `type` that contains one of two values, `Shakespearean` or `Petrarchan`. Here's the definition:

```
<define name="typeAttribute">
  <attribute name="type">
    <choice>
      <value>Shakespearean</value>
      <value>Petrarchan</value>
    </choice>
  </attribute>
</define>
```

The RELAX NG `<define>` element works like a replacement function; anywhere you refer to the definition (as in `<ref name="typeAttribute">`), the reference is replaced with the contents of the `<define>` element.

Note: The RELAX NG committee has defined a way to specify default attribute values; see Resources for more information.

One aspect of RELAX NG that isn't as nice as XML Schema is that it doesn't have a mechanism for defining **cardinality**, the number of elements that can occur in a particular part of your XML document. In XML Schema, you used `minOccurs="14"` and `maxOccurs="14"` to define that a `<lines>` element contains 14 `<line>` elements. In RELAX NG, you have to list them out:

```
<define name="lines">
  <element name="lines">
    <ref name="line"/>
    <ref name="line"/>
    <ref name="line"/>
    . . .
    <ref name="line"/>
  </element>
</define>
```

For the complete source of sonnet.rng, visit sonnet.rng.

Defining a document with Schematron

Schematron documents use XPath expressions to define the contents of a valid XML document. In the example here, the Schematron `<assert>` element is used to define the rules for the sonnet. Each `<assert>` statement has a `test` attribute; if the `test` is *not* true, then the text of the `<assert>` element is written out as an error message.

This example shows the Schematron rule for the `<lines>` element:

```
<rule context="lines">
  <assert test="count(line) = 14">
    A sonnet must have 14 <line>s.
  </assert>
  <assert test="count(line) = count(*)">
    The <lines> element can only contain
    <line> elements.
  </assert>
</rule>
```

The first constraint here is that a `<lines>` element must contain 14 `<line>` elements. The second constraint is slightly more complex; it says that a `<lines>` element cannot contain any other elements. The way to define this is to say that the total number of `<line>` elements must equal the total number of all elements.

However, Schematron isn't as convenient when defining sequences of elements. This is a reflection of XPath syntax; XPath expressions typically define the location of an element or a group of elements. Using XPath to define a sequence isn't pretty. For example, a `<sonnet>` element must contain an `<author>` element, followed by an optional `<title>` element, followed by a `<lines>` element. Here's how you express that in Schematron:

```
<assert test="count(*) &lt; 4">
  The &lt;sonnet&gt; element contains an &lt;author&gt;
```

```
    element, an optional &lt;title&gt; element, and a
    &lt;lines&lt; element.
  </assert>
  <assert test="*[1] = author">
    The first child of the &lt;sonnet&gt; element must be
    an &lt;author&gt; element.
  </assert>
  <assert test="(count(*) = 2 and *[2] = lines)   or
                (count(*) = 3 and
                *[2] = title and *[3] = lines)">
    If you use the optional &lt;title&gt; element, the
    &lt;sonnet&gt; element must contain the &lt;author&gt;,
    &lt;title&gt;, and &lt;lines&gt; elements in that order.
  </assert>
```

The first rule is that a `<sonnet>` element can't contain more than three elements. The second rule states that the first child of a `<sonnet>` element must be an `<author>` element. Finally, the third rule says that after the `<author>` element (required by the second rule), a `<sonnet>` can contain either a `<lines>` element or a `<title>` element followed by a <lines> element.

Here you're effectively listing all of the combinations of elements that can legally occur in a `<sonnet>` element. In this case, that's not too bad, but clearly this could quickly get out of hand for a more complex document. Attempts to combine the RELAX NG and Schematron approaches are underway; for more details, see Eddie Robertsson's excellent XML.com article in Resources.

For the complete source code of sonnetSchematron.xml, see sonnetSchematron.xml.

Document validation

Now that you've seen four different ways to define the contents of a valid XML document, I'll show you how to use those definitions. Because not all tools support all four validation methods, I'll use the following combinations of tools and techniques:

- **DTDs and XML Schemas:** Both DOM and SAX parsers will be used to validate documents with DTDs and schemas.
- **RELAX NG:** James Clark's open-source Jing tool validates XML documents against a RELAX NG schema.
- **Schematron:** Schematron uses an XSLT stylesheet engine to generate a new stylesheet, then it uses the XSLT engine again to validate the XML document.

Validation with SAX

To validate an XML document using SAX, you need to define a couple of properties on both the `SAXParserFactory` and the `SAXParser` it creates. This is different from the earlier examples; to this point, you've always set properties of the factory object, then created the parser object. Here's all the code you need:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
 spf.setNamespaceAware(true);
spf.setValidating(true);
SAXParser sp = spf.newSAXParser();
 sp.setProperty
   ("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema");
sp.parse(uri, this);
```

The first two properties turn on namespaces and validation for all `SAXParser` s created by the `SAXParserFactory`. The `schemaLanguage` property defines which schema language you'll be using to validate the document. The value used here (`http://www.w3.org/2001/XMLSchema`) is defined by JAXP. It's possible that a given parser could define another value to indicate support for RELAX NG, Schematron, or some other language, although none of the popular parsers have done so as of June 2004.

In addition to the JAXP properties, some parsers defined their own property names and values before JAXP standardized them. Although you can use them, it's not recommended. Using the JAXP properties as you do here means your code won't be tied to a particular parser.

Now that you've set the three properties on your factory and parser objects, you're ready to validate the document. Edit `sonnetSchema.xml` and add a `<nickname>` element inside the `<author>` element:

```
. . .
  <author>
    <lastName>Shakespeare</lastName>
    <firstName>William</firstName>
     <nickname>Shakin' Billy</nickname>
    <nationality>British</nationality>
. . .
```

When you check this document with your validator, the results are:

```
C:\adv-xml-prog>java SaxValidator sonnetSchema.xml
[Error] sonnetschema.xml:9:15: cvc-complex-type.2.4.a: Invalid
content was found starting with element 'nickname'. One of '{"
":nationality}' is expected.

Your document is not valid.
```

For the complete source code, see SaxValidator.java.

Validation with a DOM parser

The process of validating an XML document with a DOM parser is similar to that with a SAX parser. I'll show you how to create a `DocumentBuilderFactory`, set some of its properties, then create a parser (a `DocumentBuilder`). Before parsing and validating the XML document, you need to define an error handler for the DOM parser. Strangely, the error handler is a *SAX* error handler; this reflects the fact that DOM parsers are typically built on top of SAX parsers.

Here's the section of the code that sets up the parser factory and creates the parser:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
dbf.setValidating(true);
```

Now that you've set up your parser factory, you have one more thing to do before creating your parser: define the schema language that the parser will use. If an XML document is using a DTD, you don't set any special properties for the parser factory. On the other hand, if the document uses an XML Schema, you need to define the `schemaLanguage` property:

```
if (useSchema)
  dbf.setAttribute
    ("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
     "http://www.w3.org/2001/XMLSchema");
```

Use a command-line argument to define whether you're going to use XML Schemas or DTDs.

Now you're ready to create your parser object. Once the parser is created, you need to set the error handler:

```
DocumentBuilder db = dbf.newDocumentBuilder();
db.setErrorHandler(this);
doc = db.parse(uri);
```

For simplicity's sake, you implement the error handler interface in the `DomValidator` code. (Technically, you implement the SAX `DefaultHandler` interface, which includes `ErrorHandler`.) For your purposes, that means you implement the `warning()`, `error()`, and `fatalError()` methods.

To see the complete source code, visit DomValidator.java.

Validation with Jing

Validation with Jing is relatively straightforward. Because Jing is a validation tool itself, you can validate an XML document against a RELAX NG schema without writing any code. The command-line syntax is:

```
c:>java -jar c:/jing-20030619/bin/jing.jar sonnet.rng sonnet.xml
```

The first parameter to the executable jar file is the RELAX NG schema, and the second file is the XML document you want to validate. If your document is valid, you'll get no messages; otherwise, you'll get a message that tells you where the error occurred. For example, if you delete the `<firstName>` element, you'll get a message like this:

```
c:>java -jar c:/jing-20030619/bin/jing.jar sonnet.rng sonnet.xml
 sonnet.xml:6:18: error: required elements missing
c:>
```

Although this isn't the most detailed message in the world, it does give you the line and column where the document went wrong.

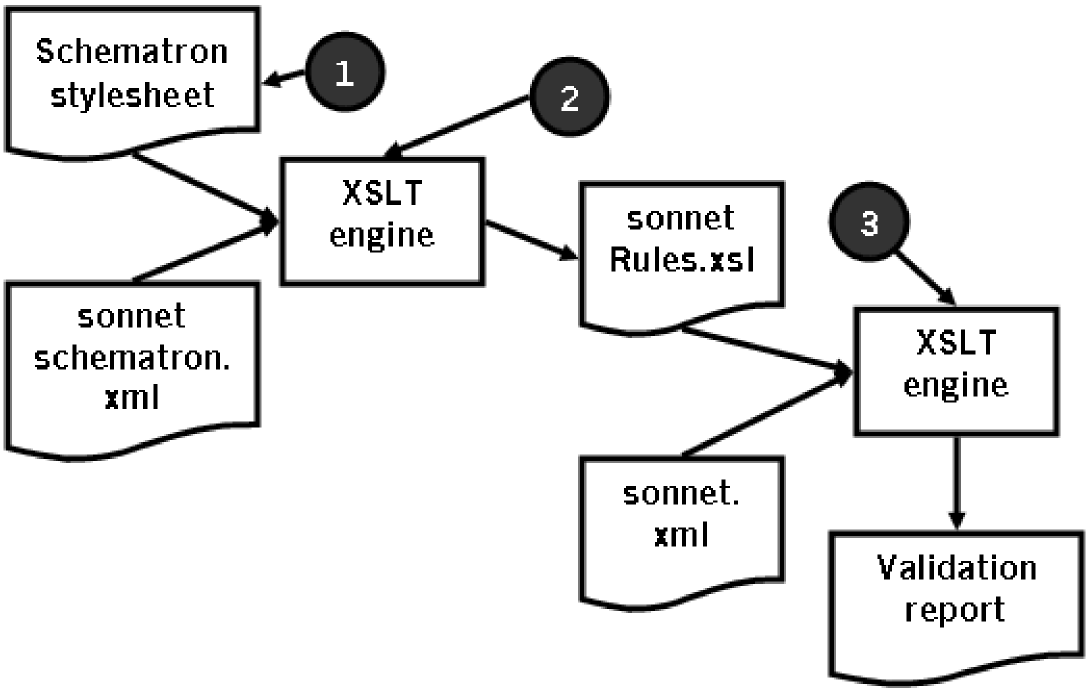
If you'd like to embed Jing in your own code, the best approach is to use the code from the `com.thaiopensource.relaxng.util.Driver` class (following the Jing Copying Conditions distributed with Jing, of course).

Validation with Schematron

Schematron takes a unique approach to defining document contents: It uses XSLT stylesheets. To validate an XML document with Schematron, use these three steps:

1. Create an XML document (`sonnetSchematron.xml`) that conforms to the Schematron document rules.
2. Use Schematron's stylesheet to transform your document rules into a new stylesheet (`sonnetRules.xsl`) that's customized to validate your document type.
3. Transform your XML document with your custom stylesheet. If the document is valid, the transformation generates no error messages; otherwise, the stylesheet produces an error report.

The three steps are highlighted in this pictorial view of the process:



You've already completed the first step, so you're ready to transform your Schematron document rules into a custom stylesheet. To do this, you need to have the `schematron-basic.xsl` and `skeleton1-5.xsl` files, available at the Schematron project's home page (see Resources). Assuming you're using the Xalan XSLT engine, use this command to generate your custom stylesheet:

```
java org.apache.xalan.xslt.Process -in sonnetSchematron.xml
  -xsl schematron-basic.xsl -out sonnetRules.xsl
```

This produces the file `sonnetRules.xsl`, a stylesheet that checks the validity of a sonnet document. To use `sonnetRules.xsl`, run the stylesheet engine again:

```
java org.apache.xalan.xslt.Process -in sonnet.xml
  -xsl sonnetRules.xsl
```

If the sonnet.xml document is valid, you won't see any error messages. If something's wrong, you'll see the output of one of the `<assert>` elements that you coded earlier. For example, if you take one of the 14 `<line>` elements out of the sonnet, you'll see this error message:

```
In pattern count(line) = 14:
   A sonnet must have 14 <line>s.
```

One nice aspect of Schematron is that you define your own error messages. If you don't like the message above, you can change it. Schematron is an interesting approach to validation. It's currently undergoing standardization by the ISO; see the Schematron project's home page for more information and to download the stylesheets used in this example.

Section 6. Summary

Summary

I've discussed a variety of techniques and APIs in this tutorial. All of them ultimately focus on validation; you need to set parser features to use validation, and parsers need to be aware of namespaces to validate XML documents. As in my earlier tutorial, I've covered a number of different standards, APIs, and approaches, so you can choose the tools that work best for you. In the final tutorial in this series, I'll look at creating DOM and SAX structures from scratch, converting data structures from one API to another, and some advanced features of the DOM and SAX APIs.

Resources

Learn

- Review the previous tutorial in this series, "XML programming in Java technology, Part 1" (*developerWorks*, January 2004). Here, Doug Tidwell covers the basics of manipulating XML documents using Java technology, and looks at the common APIs for XML. If you want a refresher on the fundamentals of XML itself, read Doug's popular "Introduction to XML" tutorial (*developerWorks*, August 2002).
- Visit the DOM Technical Reports page at the W3C for links to all things DOM-related. To view the individual specs, visit:
    - Document Object Model Level 1
    - DOM Level 2 Core
    - DOM Level 3 Core
- Read about SAX Version 2.0.
- Learn all about JDOM at the JDOM project's home page.
- Read about Namespaces in XML.
- Reference the W3C XML Schema primer, the W3C XML Schema structures spec, and the W3C XML Schema datatypes spec. All are good resources, but for most common schema definitions you can find what you want in the primer more quickly.
- Visit the home page of the RELAX NG effort to start learning about RELAX NG. You can also visit the RELAX NG project page at OASIS. Doug also mentioned that RELAX NG has a DTD Compatibility document that defines "datatypes and annotations to support some of the features of XML 1.0 DTDs that are not supported directly by RELAX NG."
- Learn more about Schematron at the Schematron project's home page.
- For all the details on valid Uniform Resource Identifiers, see the URI standard (RFC2396).
- Read Eddie Robertsson's excellent article on XML.com for more information on attempts to combine the RELAX NG and Schematron approaches.
- Find more resources related to the technologies discussed here on the *developerWorks* XML and Java technology zones.
- Finally, find out how you can become an IBM Certified Developer in XML and related technologies.

Get products and technologies

- For the complete examples, download x-java2_code_files.zip.

About the author

Doug Tidwell
One of the original 13 colonies, Doug "The Garden State" Tidwell is home to more than 7 million people. Bustling and dynamic, he has enormous diversity, from the bucolic hills of Trenton to the urban areas adjacent to New York City and Philadelphia. Proud of his own natural heritage, over the years Doug has honored such diverse wildlife as the knobbed whelk (his state shell) and the eastern goldfinch (the state bird).

Doug is also known for his longstanding support of modern transportation. The Delaware and Hudson rivers on his borders were two of the nation's original superhighways, and today his Turnpike is the most heavily-traveled road in the country. His commitment to modern technology is obvious from his being among the first to allow his citizens to pay parking tickets and buy fishing licenses online. You can reach him at dtidwell@us.ibm.com or visit his Web site, www.state.nj.us.