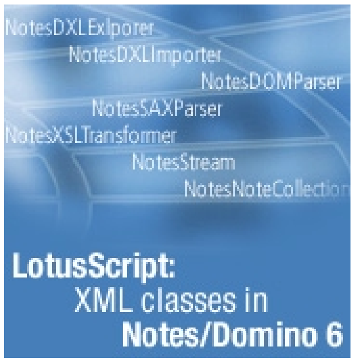


LotusScript: XML classes in Notes/Domino 6

Level: Intermediate
Works with: Notes/Domino 6
Updated: 03-Feb-2003

by Sally Blanning DeJean
and David DeJean

Notes/Domino 6 consolidates support for XML by adding several new LotusScript classes for exporting, importing, and processing XML data. This article begins a closer look at these new classes and provides some useful code examples for exporting data selectively from an NSF file in DXL format, the Domino XML language, and for applying the two parsers supported in LotusScript, the DOM parser and the SAX parser. A future article will continue with examples of applying an XSL transform to turn DXL into other XML dialects, especially HTML, and importing XML data into NSF format. (This is the second of a series of articles that take a close look at the new LotusScript classes and enhancements to the LotusScript language in Notes/Domino 6. The first dealt with the new LotusScript classes for manipulating rich text elements. See the *LDD Today* article, "**LotusScript: Rich text objects in Notes/Domino 6**.")

XML Overview

XML (which is short for Extensible Markup Language) has emerged rather quickly as an industry-standard and industrial-strength solution for exchanging data between otherwise incompatible applications and systems. XML is a markup language, like HTML: It adds tags to the content of a document. But where HTML tags are fixed and describe the appearance of the content (its typeface, size, weight, placement on the page), XML tags are unlimited and describe the structure of the content, how the pieces, or data elements, are named and how they relate to each other. In fact, XML is more than a language, it's a metalanguage—a language for creating and documenting special-purpose tag languages used to describe data structures.

Domino has provided support for XML beginning in Release 5 with the first implementation of DXL and the Lotus XML Toolkit. DXL (Domino XML Language) is an XML language especially designed to represent the structure of a Domino database written in XML format. DXL provides a tag language that describes the structure of each element—its data value, data type, attributes, and their values—and the relationship of each of these elements to the rest of the database.

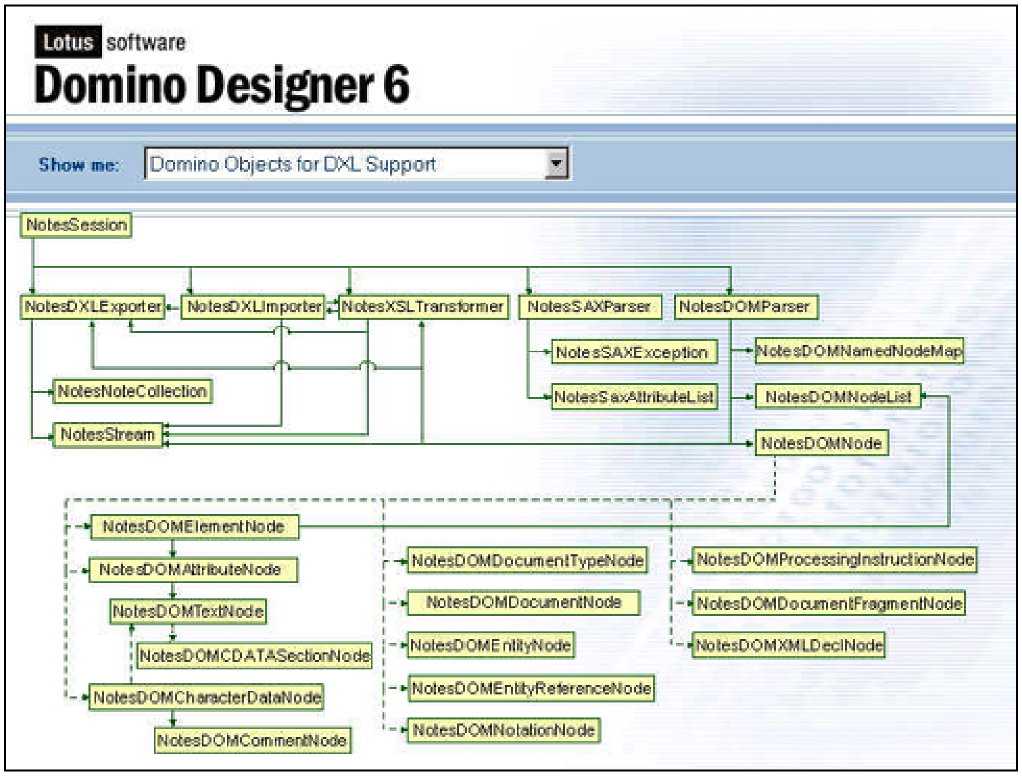
Getting a database exported as DXL, of course, is just a first step. If you want to move that data into another application, you must reformat, or transform, the data into an XML language that the receiving application understands. This translation process is performed by parsers, applications that reconfigure the tag set (and thus the structure) of the data.

The Release 5 XML Toolkit support for XML included two command-line utilities, DXLExport and DXLImport, and C++ and Java classes that implemented parsers which allowed you to get data out of an NSF file in XML format. What's new about XML support in Notes/Domino 6 is that new LotusScript classes support the full range of XML functions natively. The LotusScript implementation is more complete than the Release 5 XML Toolkit, and you can develop and debug entirely in Designer.

The core of this new support is a new base class, NotesXMLProcessor. This class contains properties and methods that are inherited by several XML processing classes. You won't use NotesXMLProcessor directly.

Instead, you create objects for one of the derived XML classes by using the appropriate Create method of the NotesSession class.

You can see a chart of the LotusScript classes for DXL by opening Domino Designer 6 to its home page and after the Show me prompt, select Domino Objects for DXL Support. This is what you'll see:



The seven most important of these classes are all new in Notes/Domino 6. They fall into three major categories:

- DXL-specific processors: NotesDXLExporter and NotesDXLImporter
- XML processors: NotesDOMParser, NotesSAXParser, and NotesXSLTransformer
- Helper classes: NotesNoteCollection and NotesStream

The DXL processor classes
The functions of NotesDXLExporter and NotesDXLImporter are straightforward: One converts Domino data to XML in the DXL language, and the other converts DXL-structured XML data to Domino data.

The XML processor classes
The three other processor classes enable Domino to perform standard XML functions.

NotesDOMParser
The XML document object model (DOM) represents data as a tree structure. All elements, or nodes, within the DOM tree may have attributes and are grouped in parent-child relationships to a root node which represents the data object as a whole. A DOM parser is an application that builds the DOM tree in memory and parses it by walking the tree from node to node. The NotesDOMParser class and its several subclasses indicated in the diagram earlier enable Domino to perform this function.

NotesSAXParser
SAX (an acronym for "Simple API for XML") is an alternative method of parsing an XML object. Rather than building a DOM tree in memory, a SAX parser reads the XML data as a stream and generates events which are processed by the application. SAX has advantages when you want to parse a relatively small amount of data out of an XML file or if the XML file is so large that building a DOM tree in memory would strain the capacity and performance of the computer running the DOM parser. The relative simplicity of the SAX approach is indicated by the diagram, but you can also see what it lacks—notably, the ability to transform the data.

NotesXSLTransformer

Almost every data export or import operation involves some filtering or transformation of the data. XML provides a way to use external documents written in XSL (XML Style Language) to control this transformation. XSL transforms can convert XML documents from one tag language to another, filter and convert using entities, and restructure the parent/child/sibling relationships in the data. XSL stylesheets are written as well-formed XML.

The helper classes
The two helper classes are particularly useful to XML processing operations, but have broader applications as well.

NotesStream
If you read the previous article in this series "**LotusScript: Rich text objects in Notes/Domino 6**" you'll recognize the NotesStream class. The NotesStream class represents a stream of binary or character data between Domino and a file or an in-memory store.

NotesNoteCollection
A Domino database consists of design and data elements known internally as notes—a document is a note as is an agent, a view, an image resource, and an ACL entry. The NotesNoteCollection class's methods and properties allow you to create an object that represents all the notes in the database or a subset selected by type and/or creation since a specified date—just the document notes, for example, or just the design notes.

The pipelining function
Pipelining is the key to working with XML in LotusScript. You can hook the processor objects and the helper objects together in assembly-line fashion so that the output of one becomes the input of the next. This technique can save you steps if you don't need to capture the interim results. You can, for example, use the NotesNoteCollection class to select documents in a database to be exported by the DXLExporter and to convert the DXL to HTML using NotesXSLTransformer.

Pipelining works because the XML processors require you to identify input and output before you call the Process method on the first object in the line. The simplest way to set up a pipeline is to specify the inputs for all the processes, but no outputs except for the output of the last process in the line.

Exporting DXL
Even if you're not familiar with XML, you'll notice right away that it looks a little bit like HTML—data is enclosed in tags. The tags can include attributes that are assigned values. They can be nested. And there's a hierarchical structure that's usually indicated by indentation.

But the tag types are nothing like HTML. HTML has a limited number of tags, but XML has an unlimited number. And where HTML tagging can be casual (you can use a <p> tag to indicate a paragraph, for instance, without bothering with the end-paragraph tag </p>), XML must be rigidly well-formed: Every opening tag, like <item>, must be matched by a closing tag, </item.> (Some tags, as you'll see in the examples that follow, are self-terminating, with a "/>" at the end.)

XML employs the tags in a schema that describes the data. DXL is one such schema which describes a Domino database. Tags like <databaseinfo> and <aclentry> are descriptors for pieces of the database structure. The DXL schema includes descriptors for all the pieces of an NSF file—documents, forms, views, agents, ACL, and much, much more. A DXL file represents the contents of the database, all properly tagged, in plain text. Here's the top of a DXL file:

```
<?xml version='1.0' encoding='utf-8'?>
    <!DOCTYPE database SYSTEM 'xmlschemas/domino_6_0.dtd'>
        <database xmlns='http://www.lotus.com/dxl' version='6.0' replicaid='85256C7500771804'
        path='dxlhelloworld.nsf' title='DXL Hello World'>
        <databaseinfo dbid='85256C7500771804' odsversion='43' diskspace='327680' percentused='86.40625'
        numberofdocuments='1'>
            <datamodified><datetime>20021118T165757,31-05</datetime></datamodified>
            <designmodified><datetime>20021118T165803,33-05</datetime></designmodified>
        </databaseinfo>
        <acl maxinternetaccess='editor'>
            <aclentry name='-Default-' default='true' level='noaccess' readpublicdocs='true'
            writepublicdocs='true'/>
            <aclentry name='OtherDomainServers' type='servergroup' level='noaccess' readpublicdocs='false'
            writepublicdocs='false'/>
```

Whatever the target format, every export operation begins with DXL. At its simplest, it may end there as well. Here's a very simple agent that exports the contents of a database as DXL. It's part of the database named DXL Hello World included in the zip file of examples for this article that you can download from the **Sandbox**. The database contains one view, one form, and one document with one data field. Before you open the DXL Hello World Database, create a directory on your hard disk with the path name C:\dxl. Then open the database, open the single document, and choose Actions - 1. Export DXL to run the agent.

The Export DXL agent
The agent, adapted with slight changes from the Domino Designer Help file, uses a NotesStream object to write the contents of the NSF to a file named after the database—dxlhelloworld_all.xml. We could have given the file a DXL extension, which would perhaps have been more specific—it would help make clear that the file is in DXL format. We chose an XML extension to take advantage of Internet Explorer's automatic association with the XML file type and ability to render XML files into a nicely formatted display. If you double-click on the file in Windows Explorer, it will open in IE—if it's a valid XML file. (If it's not, you'll get an error message and a pointer to the location of the problem in the file. This can help you debug problems when creating XML files.)

The first part of the Export DXL agent creates a NotesStream object to represent the contents of dxlhellowworld.nsf:

```
Sub Initialize
    Dim session As New NotesSession
    Dim db As NotesDatabase
    Set db = session.CurrentDatabase
    Dim stream As NotesStream
    Set stream = session.CreateStream
    filename$ = "c:\dxl\" & Left(db.FileName, Len(db.FileName) - 4) & "_all.dxl"
    If Not stream.Open(filename$) Then
        Messagebox "Cannot open " & filename$ & ". Check to make sure this directory exists.",, "Error"
        Exit Sub
    End If
    Call stream.Truncate
```

(The NotesStream.Truncate method raises an error if the output file is read-only and cannot be written to.)

Next the NotesDXLExporter object is set with two parameters, input and output, and the process is called. Note the call that actually executes the exporter process is separate from the specification of the exporter object. This becomes important when pipelining is involved, and there's quite a bit more set-up to do before the processing:

```
    Dim exporter As NotesDXLExporter
    Set exporter = session.CreateDXLExporter(db, stream)
    exporter.OutputDOCTYPE = False
    Call exporter.Process
End Sub
```

If you don't set the input and output in the CreateDXLExporter method, you can do it in the SetInput and SetOutput methods, a useful technique if you're iterating export operations. Here's how that would look:

```
    Dim exporter As NotesDXLExporter
    Set exporter = session.CreateDXLExporter
    Call exporter.SetInput(db)
    Call exporter.SetOutput(stream)
    exporter.OutputDOCTYPE = False
    Call exporter.Process
End Sub
```

The line exporter.OutputDOCTYPE=False raises some deeper issues about XML. If this line were set to True (or omitted because True is the default), the output would include a line that says:

```
<!DOCTYPE database SYSTEM 'xmlschemas/domino_6_0.dtd'>
```

This line, called the DOCTYPE declaration, points to a file that defines the structure of the XML document. This document type definition, or DTD, declares all of the document's element types, children element types and their order and number, and all the HYPERLINK "http://xmlwriter.net/xml_guide/attlist_declaration.shtml" attributes, entities, and other parts of the document. The Domino DOCTYPE declaration, for example, declares that the database element is the root element of the XML DOM tree created by the document and that the other declarations are contained in a file on the same SYSTEM as the document with the path name xmlschemas/domino_6_0.dtd. Indeed, if you look in your Notes directory, there beside the Data and JVM and MUI folders, you will see an XMLSchemas folder that contains a 147Kb DTD file, domino_6_0.dtd, which you can open and read.

What is a DOCTYPE?
So why have—or not have—a DOCTYPE declaration? There are many mysteries of XML, XSL, DOM, and SAX that are beyond the scope of this article, and we're getting perilously close to some of them here. Suffice to say that there are two kinds of XML—valid and well-formed. Valid XML has a DTD. Well-formed XML has no DTD, but is properly constructed—all entities are declared, properly closed, and correctly nested. The XML specification makes a DTD optional. Well-formed XML is self-documenting and can be processed without a DTD. Valid XML cannot. Applications like Internet Explorer that don't know how to use the DTD choke on the DOCTYPE declaration and refuse to process the file.

Because some applications that might use Domino DXL files depend on there being a DOCTYPE declaration and some depend on there not being a DOCTYPE declaration, the XML specification makes it optional and the NotesDXLExporter class supports this with the OutputDOCTYPE property.

Actually, in this agent, it doesn't make any functional difference because we're not doing anything with the output. But because we want to view the output in Internet Explorer, it's set to False. If we were pipelining the DXL to another XML process, we would want it to be set to False, as well. In general, omitting a DOCTYPE declaration from the output is a good starting point.

Using NotesNoteCollection

The output of the Export DXL agent is a 40 KB file. Because it does not include a DOCTYPE declaration you can open in Internet Explorer. When you do that, what you'll see will look familiar—it starts with the same DXL code used as an example at the start of this article (minus the DOCTYPE, of course). As you read through it, you'll recognize all the pieces of an NSF file—ACL, log entries, agents, even the database icon is represented.

For a database that contains only one document, 40 KB of data is probably more than you need to deal with. Unless your goal is to eventually recreate the NSF file, you don't really want to export the design notes, ACL entries, and so forth. The place to start winnowing out items you don't want is with the NotesNoteCollection.

The NotesNoteCollection class includes properties that let you toggle the types of notes you want to include in the export. Some 30 individual note types can be specified as properties—SelectDocuments, SelectHelpAbout, SelectSubforms, and so on. In addition, half a dozen methods let you include or exclude all notes of several related types. The SelectAllCodeElements method, for example, is one-stop shopping for all the agents, database scripts, script libraries, data connections, outlines, and miscellaneous code elements in the database. The properties and methods are turned on and off by Boolean arguments—true (that is, notes of this type will be exported) or false (notes of this type will not be exported). In addition, a Boolean property of the CreateNoteCollection method acts as a master switch. You can set CreateNoteCollection(false) to start your selection from zero, or CreateNoteCollection(true) to start with everything, then use the property for each note type or one of the methods that encompass several types to tailor your selection. For instance:

```
Dim nc as NotesNoteCollection
Set nc = db.CreateNoteCollection(False)
Call nc.SelectAllDesignElements(True)
```

would create a NoteCollection object that included only design elements, while

```
Dim nc as NotesNoteCollection
Set nc = db.CreateNoteCollection(True)
Call nc.SelectAllDesignElements(False)
```

would create a NoteCollection object that included all the notes from the database except the design elements. Once you have set these constructor properties, the collection object is created by calling the BuildCollection method. This is illustrated in the next example.

The Export Only Documents agent

To create a NoteCollection object that contains only documents and omits all the other database pieces, run the 2. Export Only Documents agent in the DXL Hello World database. This agent uses the SelectDocuments property of NotesNoteCollection:

```
Dim nc As NotesNoteCollection
Set nc = db.CreateNoteCollection(False)
nc.SelectDocuments=True
Call nc.BuildCollection
```

The NotesNoteCollection object is specified as the input for the DXLExporter:

```
Dim exporter As NotesDXLExporter
Set exporter = session.CreateDXLExporter(nc, stream)
exporter.OutputDOCTYPE = False
Call exporter.Process
```

The result of this agent is an XML file named dxlhelloworld_documents.xml that is much briefer because it contains only the elements that represent document notes (in this case, just one note), the root element for the file, named <database>, that contains the document elements, and of course the elements contained by the document element. It is, in fact, brief enough to be included here:

```
<?xml version='1.0' encoding='utf-8'?>
    <!DOCTYPE database SYSTEM 'xmlschemas/domino_6_0.dtd'>
        <database xmlns='http://www.lotus.com/dxl' version='6.0' replicaid='85256C7500771804'
        path='dxlhelloworld.nsf' title='DXL Hello World'>
            <databaseinfo dbid='85256C7500771804' odsversion='43' diskspace='327680'
            percentused='88.828125' numberofdocuments='1'>
                <datamodified>
                    <datetime>20021118T165757,31-05</datetime>
                </datamodified>
                <designmodified>
                    <datetime>20021125T165404,83-05</datetime>
                </designmodified>
            </databaseinfo>
            <document form='Hello'>
                <noteinfo noteid='8fa' unid='7C962ABED6913FAD85256C750078A86F' sequence='1'>
                <created>
                    <datetime>20021118T165754,39-05</datetime>
                </created>
                <modified>
                    <datetime>20021118T165757,31-05</datetime>
                </modified>
                <revised>
                    <datetime>20021118T165757,30-05</datetime>
                </revised>
                <lastaccessed>
                    <datetime>20021118T165757,30-05</datetime>
                </lastaccessed>
                <addedtofile>
                    <datetime>20021118T165757,30-05</datetime>
                </addedtofile>
            </noteinfo>
            <updatedby>
                <name>CN=David DeJean/O=DeJean</name>
            </updatedby>
            <item name='HelloData'>
                <text>Hello World.</text>
            </item>
        </document>
    </database>
```

This output probably is still more verbose than you want. But to go any further, you'll have to not just omit elements, but also edit the structure of the DXL file. And to do that, you'll have to parse the data with one of the three XML parsing tools—the DOM parser, the SAX parser, or the XSL transformer—that are newly available to LotusScript. In this article, we'll look at the DOM and SAX parsers, but we'll save the XML transformer for the next article in this series.

Using the SAX parser

The NotesNoteCollection class is a relatively inflexible programming tool. You can choose the notes you want to include in your DXL output and the notes you want to ignore, but you cannot alter the structure and content of the notes themselves, which is determined by the Domino DTD. A DXL file includes not only all the data that was saved in the Domino database, but all the meta-data—when that information was saved and modified, and who saved it, and how it should be presented.

This meta-data can be overwhelming. In the previous example, the data amounts to "Hello World." It is set in—or perhaps concealed by—no fewer than 26 items of meta-data: database name and attributes, databaseinfo and attributes, document name and attributes, and item name. There are seven date/time values, two data types, three unique identifiers, and statistics on how much disk space the original database occupied.

Much of this is unnecessary for any purpose outside of Domino. And yet there it is, often nested several layers deep inside an element you need to keep. The elements named <databaseinfo>, <noteinfo>, and <updatedby>, for example, are all contained by the database element. Their values are nested within elements that serve as datatype identifiers—datetime, text. (The original XML specification didn't recognize data types and treated everything as text.)

What's the solution? If the DXL transporter could look at each element within a note and decide whether to ignore it or transfer it to the DXL output file that might help, but it would make the transporter much bigger and besides, there are XML processes that already do that: They're called parsers. The DOM parser and the SAX parser use different data models.

The Export Only Data—SAX agent

The Domino SAX parser can raise a baker's dozen events: SAX_Characters, SAX_EndDocument, SAX_EndElement, SAX_Error, SAX_FatalError, SAX_IgnorableWhitespace, SAX_NotationDecl, SAX_ProcessingInstruction, SAX_ResolveEntity, SAX_StartDocument, SAX_StartElement, SAX_UnparsedEntityDecl, SAX_Warning. LotusScript can respond by hooking each event to a different subroutine. To see an example agent, look at the one named 3. Export Only Data—SAX in the DXL Hello World database. When you run this agent from the Actions menu, it creates a file named dxlhelloworld_sax.xml in the C:\dxl directory.

The task for this SAX parser agent is to cut the meta-data down to size and output an XML file that contains the real data in the DXL Hello World database and as little of everything else as possible.

The code begins just like the Export Only Documents script: It creates a NotesNoteCollection that is only documents and exports the collection as DXL. Then it pipelines the DXL to a SAX parser. The pipelining is automatic—the DXL exporter has an input argument (nc) but no output argument. (If there were other processor objects in the pipeline, they would likewise specify their input, but not their output.) The SAX parser, as the last process in the pipeline, has arguments for both its input, the DXL exporter, and its output, a NotesStream object:

```
Sub Initialize
	Dim session As New NotesSession
	Dim db As NotesDatabase
	Set db = session.CurrentDatabase

'Build a NoteCollection to limit the export file to documents
	Dim nc As NotesNoteCollection
	Set nc = db.CreateNoteCollection(False)
	nc.SelectDocuments=True
	Call nc.BuildCollection

'Create the DXL exporter
	Dim exporter As NotesDXLExporter
	Set exporter = session.CreateDXLExporter(nc)
	exporter.OutputDOCTYPE = False

'Create the output file
```

```
    Dim xml_out As NotesStream
    Set xml_out=session.CreateStream
    filename$ = "c:\dxl\" + Left(db.FileName, Len(db.FileName) - 4) + "_sax.xml"
    If Not xml_out.Open(filename$) Then
        Messagebox "Cannot open " + filename$ + ". Check to make sure this directory exists.",, "Error"
        Exit Sub
    End If
    Call xml_out.Truncate

'Create the SAX parser
    Dim saxParser As NotesSAXParser
    Set saxParser=session.CreateSAXParser(exporter, xml_out)
```

The remainder of the agent defines the connections between each SAX parser event and a matching subroutine:

```
    On Event SAX_Characters From saxParser Call SAXCharacters
    On Event SAX_EndDocument From saxParser Call SAXEndDocument
    On Event SAX_EndElement From saxParser Call SAXEndElement
    On Event SAX_Error From saxParser Call SAXError
    On Event SAX_FatalError From saxParser Call SAXFatalError
    On Event SAX_IgnorableWhitespace From saxParser Call SAXIgnorableWhitespace
    On Event SAX_NotationDecl From saxParser Call SAXNotationDecl
    On Event SAX_ProcessingInstruction From saxParser Call SAXProcessingInstruction
    On Event SAX_StartDocument From saxParser Call SAXStartDocument
    On Event SAX_StartElement From saxParser Call SAXStartElement
    On Event SAX_UnparsedEntityDecl From saxParser Call SAXUnparsedEntityDecl
    On Event SAX_Warning From saxParser Call SAXWarning

    exporter.Process
End Sub
```

(Notice that the line that initiates processing says exporter.Process, not saxParser.Process. When you use pipelining, you set up all the process objects, then start processing by calling the first object in the pipeline.)

You don't need to write a handler subroutine for each event. This example agent has been adapted from an agent in the Domino Designer help file that puts up a message box for each SAX event. The complete list gives you a starting point for your own code. You may not care about ignorable white space or entities. (You probably want to know about errors and warnings, however.)

Each of these event handlers queries the data through the parameters and must explicitly generate any output with the Output method. The handler for character data, for example, ignores nulls, and writes anything else to the output file as a string:

```
Sub SAXCharacters (Source As Notessaxparser, Byval Characters As String,_
Count As Long)
    If Characters <> Chr(10) Then
        Source.Output(Characters)
    End If
End Sub
```

With each of the handlers formatting whatever output is required, the Export Only Data—SAX agent parses an XML file and outputs another XML file. When the SAXStartDocument subroutine is triggered. it writes an XML declaration and a linefeed/carriage return to the output file:

```
Sub SAXStartDocument (Source As Notessaxparser)
    Source.Output({<?xml version='1.0' encoding='utf-8'?>} + Chr(13)+Chr(10))
End Sub
```

The SAXStartElement subroutine formats the start of an element and its attributes, if any. It includes three If statements that check for particular element names and that exit the subroutine without writing the element to the output if they are found:

```
Sub SAXStartElement (Source As Notessaxparser, Byval ElementName As String, Attributes As
NotesSaxAttributeList)

    If ElementName = "databaseinfo" Then
        Exit Sub
    End If
    If ElementName = "noteinfo" Then
        Exit Sub
    End If
    If ElementName = "updatedby" Then
        Exit Sub
    End If

    Source.Output({<}+ ElementName)
```

The attributes are handled using the NotesSAXAttributesList class, an array automatically created by the parser that contains the names and values of all the element's attributes. A loop selects each attribute in turn, and the Source.Output() statement writes the attribute name and value and formatting characters to the output. When all the attributes are processed, the subroutine closes the element start tag with a ">" and exits:

```
Dim i As Integer
    If Attributes.Length > 0 Then
        Dim attrname As String
        For i = 1 To Attributes.Length
            attrname = Attributes.GetName(i)
            Source.Output({ } + attrname+{="}+Attributes.GetValue(attrname) + {"})
        Next
    End If
    Source.Output({>})
End Sub
```

If the element has a value, it will appear next as character data, and the SAX parser will raise a SAX_Characters event. The SAXCharacters subroutine checks to make sure that the entire content of the value isn't a Chr$(10), and if it's something more, writes it to the output:

```
Sub SAXCharacters (Source As Notessaxparser, Byval Characters As String,_
Count As Long)
    If Characters <> Chr(10) Then
        Source.Output(Characters)
    End If
End Sub
```

The SAX_EndElement event triggers the SAXEndElement subroutine, which contains checks for the three element names that were dropped by the SAXStartElement subroutine. If the element name is something else, it writes a formatted close tag and linefeed/carriage return to the output:

```
Sub SAXEndElement (Source As Notessaxparser, Byval ElementName As String)

    If ElementName = "databaseinfo" Then
        Exit Sub
    End If
    If ElementName = "noteinfo" Then
        Exit Sub
    End If
    If ElementName = "updatedby" Then
        Exit Sub
    End If

    Source.Output(
</} + ElementName + {>} + Chr(13)+Chr(10))
End Sub
```

The output of this agent looks a lot like the output of the Export Only Documents agent—a little too much, in fact. Here's the output as formatted by Internet Explorer for display:

```
<?xml version="1.0" encoding="utf-8" ?>
    HYPERLINK "C:\dxl\" - <database xmlns="http://www.lotus.com/dxl" version="6.0"
    replicaid="85256C7500771804" path="dxlhelloworld.nsf" title="DXL Hello World">
    HYPERLINK "C:\dxl\" - <datamodified>
        <datetime>20021209T173711,55-05</datetime>
    </datamodified>
    HYPERLINK "C:\dxl\" - <designmodified>
        <datetime>20021210T132250,26-05</datetime>
    </designmodified>
    HYPERLINK "C:\dxl\" - <document form="Hello">
    HYPERLINK "C:\dxl\" - <created>
        <datetime>20021204T092656,26-05</datetime>
    </created>
    HYPERLINK "C:\dxl\" - <modified>
        <datetime>20021204T092658,78-05</datetime>
    </modified>
    HYPERLINK "C:\dxl\" - <revised>
        <datetime>20021204T092658,77-05</datetime>
    </revised>
    HYPERLINK "C:\dxl\" - <lastaccessed>
        <datetime>20021204T092658,77-05</datetime>
    </lastaccessed>
    HYPERLINK "C:\dxl\" - <addedtofile>
        <datetime>20021204T092658,77-05</datetime>
    </addedtofile>
    <name>CN=David DeJean/O=DeJean</name>
    HYPERLINK "C:\dxl\" - <item name="HelloData">
        <text>Hello World.</text>
    </item>
    </document>
    </database>
```

We successfully removed the elements named <databaseinfo>, <noteinfo>, and <updatedby> and all their attributes, but their subelements are still here. The elements <datamodified> and <designmodified>, which were children of <databaseinfo>, are now children of <database>. The five date/time elements that were children of <noteinfo> are now children of <document>. The <name> element formerly a child of <modifiedby> is now a child of <document> as well. What happened?

The SAX parser did exactly what we told it to—it eliminated start tags and end tags for the three elements we named. That's all it could do because that's all it knows how to do. It doesn't know children or subelements. It just knows events. For many purposes, and with many XML files, this would be enough—easy to write, easy to run. But the complexity of the Domino DTD needs something stronger. Fortunately, we've got it.

Using the DOM parser

The NotesDOMParser class implements a DOM parser in LotusScript. The code that invokes it looks a lot like the other NotesXMLProcessor objects, no surprise. But a DOM parser functions very differently from a SAX parser. The DOM tree this parser builds is a memory object, not just a stream of events. The parser then traverses this tree recursively. The processing isn't limited to one shot at whatever data is represented by a SAX event. Control of the output can be related to the existence and contents of other nodes in the tree.

In DOM terminology, each item in this tree is a node and has its own properties. The LotusScript class named NotesDOMNode provides a property, NodeType, used to identify the type of the current node. Classes derived from NotesDOMNode represent the various types of nodes: element nodes, attribute nodes, text nodes, and special-purpose nodes like the XML declaration node. These classes let LotusScript code handle each instance of a node type. Instead of all the event handlers that are set for the SAX parser, the DOM parser needs only one subroutine that traverses the tree, checks the type of node it finds, and takes whatever action you've programmed using the appropriate class.

The Export Only Data—DOM agent

The agent named 4. Export Only Data—DOM is written to tackle the same problem we set for the SAX parser agent—to separate the data from as much of the meta-data as possible. When you look at the agent code, you'll see that it begins just like SAX parser script: It creates a NotesNoteCollection that is only documents and exports the collection as DXL. Then it pipelines the DXL; this time to a DOM parser. The pipelining works the same way: The DXL exporter has an input argument (nc) but no output, and exporter.Process starts the processing. The DOM parser takes the output from the exporter and turns into the DOM tree. The subroutine, called walkTree, traverses the tree:

```
    Dim docNode As NotesDOMDocumentNode
Set docNode = domParser.Document
    Call walkTree(docNode)
```

The Document property of the NotesDOMParser object represents the root node of the DOM tree. The walkTree subroutine traverses the tree beginning at this root. After some set-up, the subroutine begins with a Select Case statement. In this statement, the object named node is an instance of NotesDOMNode. The capitalized values in the case statements that follow are legal values for the NodeType property. When there's a match, the code contained in the matched case is executed:

```
Sub walkTree (node As NotesDOMNode)
...
If Not node.IsNull Then
Select Case node.NodeType
```

In fact, there's an immediate match to DOMNODETYPE_DOCUMENT_NODE because every DOM tree has a Document node, so the subroutine does three things. First, the basics: It gets the first child node of Document and the total number of child nodes, so that it can traverse the next level of the tree:

```
    Case DOMNODETYPE_DOCUMENT_NODE:    'It's the Document node
    'Get the number of children of Document
        Set child = node.FirstChild    'Get first child node
        Dim numChildNodes As Integer
        numChildNodes = node.NumberOfChildNodes
```

Next, because we're formatting the output as XML, we write a very simple, but very necessary XML declaration to the output using properties of the class NotesDOMXMLDeclNode, basing it on the properties of the first child node of Document, which must always be an XML declaration:

```
    'Create an XML declaration for the output
        Dim xNode As NotesDOMXMLDeclNode
        Set xNode = node.FirstCild
        domParser.Output({<?xml version="} + xNode.Version  + {" ?>})
```

Finally, the subroutine gets the next node and recurses—it calls itself with the object named child, now the second child node of Document, as an argument. It does this inside a While statement, decrementing the number of unprocessed child nodes as it iterates:

```
    'Call walkTree for the first child
        While numChildNodes > 0
            Set child = child.NextSibling    'Get next node
            numChildNodes = numChildNodes - 1
            Call walkTree(child)
        Wend
```

Only two of the other cases are significant for our agent. If the current node is a text node with a value of anything but a linefeed, the value is written to the output (the elimination of these characters is purely cosmetic—they come from a few otherwise empty fields in the Domino data and appear as stray box characters when the output is rendered in Internet Explorer, so we zap them here:

```
    Case DOMNODETYPE_TEXT_NODE:    'It's a plain text node
        If node.NodeValue <> Chr(10) Then
            domParser.Output(node.NodeValue)
        End If
```

The core node type is the element note. Most nodes in the tree are of this type; they can have both attributes and values. When this case is matched, the first thing we do is the same thing we did in the SAX Parser StartSAXElement subroutine; we check for the three element names we want to eliminate and exit the subroutine without writing to the output if we find them. If the element isn't one of the three, we write the first part of the element tag, a "<", and the element's name, to the output:

```
    Case DOMNODETYPE_ELEMENT_NODE:
        If node.NodeName = "databaseinfo" Then
            Exit Sub
        End If
        If node.NodeName = "noteinfo" Then
            Exit Sub
        End IfIf node.NodeName = "updatedby" Then
            Exit Sub
        End If
        domParser.Output({<} + node.NodeName)
```

Next we check for attributes, and if the element has any, we write them to the output with proper formatting and then close the tag with a ">":

```
        Dim numAttributes As Integer, numChildren As Integer
        numAttributes = node.attributes.numberofentries
        Set attrs = node.Attributes    'Get attributes

        Dim i As Integer
        For i = 1 To numAttributes    'Loop through attributes
            Set a = attrs.GetItem(i)
            domParser.Output({ }+a.NodeName + {="} + a.NodeValue + {"})
        Next
        domParser.Output(">")
```

We check for child nodes, and if any exist, we get the first one and recurse again through walkTree using a While statement just as we did before:

```
        numChildren =  node.NumberOfChildNodes
        Set child = node.FirstChild    'Get child
        While numChildren > 0
            Call walkTree(child)
            Set child = child.NextSibling    'Get next child
            numChildren = numChildren - 1
        Wend
```

Finally, when all the children of the current node have been processed and we drop out of the While statement, we write the closing tag for the element:

```
        domParser.Output( {</} + node.nodeName + {>} + LF)
```

When all the nodes in the tree have been processed, control returns to the main Initialize subroutine at the label results. The output stream is closed and we're all done:

```
results:
    Call outputStream.Close
    Exit Sub
errh:
    outputStream.WriteText ("errh: "+Cstr(Err)+":  "+Error+LF)
    Resume results
End Sub
```

The label errh is an important part of this code as well. If an error had occurred at any point, the error number and a message would have been written as the output file and the parser would have quit. XML parsers are not tolerant of errors because the likeliest errors involve data that is not well formed. If the data is not well formed, it's not XML, so it's not processed. Contrast this to Web browsers, for example, which are HTML parsers. Web browsers are fault-tolerant in the extreme because with HTML the goal is to render something to the screen, even if it's not an entirely accurate representation of the code. XML, on the other hand, sets consistency above all.

The output of the DOM parser is significantly more compact than what we got from the SAX parser:

```
<?xml version="1.0" ?>
    <database path="dxlhelloworld.nsf" title="DXL Hello World" xmlns="http://www.lotus.com/dxl" version="6.0"
    replicaid="85256C7500771804">
        <document form="Hello">
            <item name="HelloData">
                <text>Hello World.</text>
            </item>
        </document>
    </database>
```

It's well formed XML. It doesn't include the three element types we wanted removed. And more importantly, all of the child date/time and <name> elements the SAX parser passed to the output have been eliminated as well. The reason is the recursion: When we stopped the processing of the node, we automatically stopped processing of all its child nodes as well. The SAX parser, in contrast, raised events for all these child elements regardless of whether their parents had been processed. We could have improved the SAX parser output if we had written more code, but we would have had to add 10 more If statements to check for item names we wanted to eliminate. The DOM parser suited our purpose better. In another situation, it might not have. There are no hard and fast rules on when to choose one over the other—or either over the XSL transformer, which we'll look at closely in the next article in this series.

Conclusion

This article gives you an introduction to XML and the DOM and SAX parsers, though there is much more to be said about Domino's support for XML. The examples presented here should get you started importing and exporting data as DXL and should give you some ideas for how to use LotusScript to manipulate XML. In the next article in this series, we'll look at extensible style sheets (XSL) for transforming DXL into other formats, including HTML and other XML languages.

ABOUT THE AUTHORS

Sally Blanning DeJean and David DeJean have been working with and writing about Lotus Notes and Domino for as long as these products have existed. They were co-authors of the very first book about Notes, *Lotus Notes at Work.* Sally, a CLP Principal, has written other books about Notes and is a full-time Notes/Domino developer. David, a CLP, has been an editor and writer for several computer publications. He is a partner in DeJean & Clemens, a firm that develops Notes and Internet applications and technical and marketing communications.