

Introducing the Document Object Model using OpenXML (Part I)

by Craig Murphy

Need native access to XML documents? Don't want to have heavy client installs that include an XML parser and an Internet browser? OpenXML is a native Delphi solution that sports access to in-memory XML documents via a Document Object Model (DOM) – without any excess baggage.

Introduction

Those of you who know me will realise that the eXtensible Markup Language (XML) is my “thing”. Much of my evangelising about XML has revolved around using the Microsoft XML parser that comes with Internet Explorer 5.x – known as MSXML. Whilst this parser is very good (in my opinion), using it necessitates that IE is installed on all the machines on which you plan to deploy your application. This may prove to be a problem for some of your clients, especially those who have rolled out browsers such as Netscape. OpenXML alleviates this problem by offering two components that allow you to implement XML-solutions that rely on nothing more than the executable that makes up your application. It gets better – the solutions that you build with OpenXML can easily be re-compiled using Kylix, thus you are [nearly] able to build single-source applications.

Over the course of this article, I show you how to integrate XML into your applications using the OpenXML implementation. There's a lot to cover, OpenXML implements the World Wide Web Consortium (W3C) Document Object Model (DOM), so this is the first of a two-part article.

Wherever we turn these days, somebody (myself included!) is ranting on about XML and how it is the panacea of Business-2-Business (B2B) data interchange. The same people also go on about how XML can be used for Application-2-Application (A2A) communication and even inter-object or Object-2-Object (O2O). So, XML must have some award-winning attributes.

XML in a paragraph (nearly)

I'm sure we're all familiar with XML however, to ensure that we're all singing from the same hymn sheet, I'll cover the salient points here. XML documents are text-based and are therefore platform independent. Listing 1 presents a very small XML document that demonstrates some of the XML syntax.

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!-- Simple message -->
3: <richplum display="true">
4: <message>hello, world</message>
5: </richplum>
```

Listing 1

Line 1 is known as a **processing instruction**. Line 2 is a comment. Line 3 is the **root node** (or document element) – the root node <richplum> is a single node (or element) that is said to contain zero or more child nodes. Nodes have a start-tag (<richplum>) and an end-tag (</richplum>). Associated with the <richplum> node is an **attribute** called

display. Attributes are always delimited by quotation marks. Line 4 contains the <message> node; the content of this node is the string “hello, world”. Line 5 is the end-tag for the document element.

As a short aside, how do we know that the <message> node contains something of type string? Equally how do we know that the display attribute is a Boolean? Well, on the surface, we don't know – because type information is missing. We can add type information to XML elements using XML Schema [1]. A detailed examination of XML Schema is beyond the scope of this article – however you may wish to investigate it on your own.

We are able to work with XML documents using an XML Parser. OpenXML, like MSXML, is an XML Parser – it's capable of parsing through the XML grammar, character by character. Once an XML document has been parsed (either in part or completely) we need a mechanism that allows us to work with the XML.

Working with XML documents

Parsing XML documents is performed using two distinct methods. The first method is known as the Simple API for XML (SAX). SAX involves accessing an XML document incrementally, in an event driven fashion – the XML document is not loaded into memory – it's read from disk in a forward-only manner. The second method is known as the Document Object Model (DOM) – this involves loading the entire XML document into an object. SAX is very memory efficient, whereas the DOM is not.

The inefficiencies of the DOM are only apparent if you expect the DOM to handle very large XML documents (large being greater than 1MB, in my opinion). Some DOM implementations load the XML document into a memory space that expands to four times the size of the original XML document. Using the DOM to access XML documents effectively allows random access to the entire XML document. OpenXML offers access to XML documents via a DOM.

The DOM is simply a set of standard interfaces; a DOM by itself does nothing more than provide a specification that a DOM implementation (such as OpenXML) should adhere to (implement). To avoid confusion with the HTML DOM – as used by most web browsers, you'll often see reference to an XML DOM, e.g. “...just load the employee list into an XML DOM...”

XML documents shouldn't be used for long-term storage of information that you would normally hold in a database. DOM-based XML implementations excel with small XML fragments. Since most XML documents that you'll create are the result a query on a database, they should be fairly small anyway. Equally, any XML documents that you need to send between applications or businesses should contain only the information that is required – all too often SQL queries are unoptimised, e.g. `SELECT * FROM ...` whereas a `SELECT EMP_NO, EMP_LASTNAME` is more optimal. Using XML for long-term storage of small inifile-like system configuration information is also practical – it was this sort of problem that made me sit down and use OpenXML.

What's in the box?

So, you've downloaded OpenXML[2] from the URL given in the Resources section of this article – let's go through the bits. `XDOM_2_3.pas` contains nearly 23,000 lines of Pascal and provides the source for eight components. The two primary components that make up OpenXML are: `TDomImplementation` and `TXmlToDomParser`. Thankfully, we don't need to understand every line of code – using OpenXML requires only a few lines of code. The OpenXML author provides a detailed manual – supplied as an XML document, of course! You do need Internet Explorer to read it – however there is an HTML Help file available too. The manual itself is largely orientated around the source code. This can be a good thing, but can make the novice think twice. Similarly, there are very few examples provided with the download – a gap that I am hoping to fill here, and one that Charlie Calvert[3] has also touched on.

Whilst there are many DOM implementations available, few of them can be considered 100% faithful to the W3C DOM[4] specifications. For example, the DOM specification doesn't actually provide an interface that allows an XML document to be loaded from disk. Most of the implementations that I've looked at have such a method: MSXML uses the `load` method; OpenXML uses a `filetodom` method. Therefore, DOM implementations are said to extend the specification. DOM implementations often extend the specification in other areas too – OpenXML is no exception. As the OpenXML DOM implementation extends the DOM specification the OpenXML DOM is known as the eXtended DOM (XDOM).

An Introduction to the DOM

The DOM represents XML using what is known as a DOM node tree. If we load listing 2 into an XML DOM, we would have a DOM node tree similar to Figure 1. The DOM treats everything as nodes (or elements), even attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<richplum display="true">
<message>hello, world</message>
</richplum>
```

Listing 2

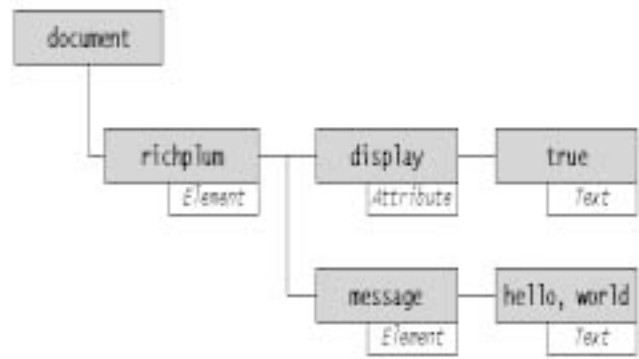


Figure 1 How listing 2 is represented using a DOM tree

Looking at Figure 1, we can see that each item in the DOM tree has a type: Element, Attribute, or Text. The DOM specification defines a number of node types. We take a closer look at node types in the second part of this article.

Building a DOM tree

I've created a form with three components: a button, a memo and a `TDomImplementation`. Listing 3 presents the code that is attached to the button. This demonstrates how to create an in-memory XDOM, and then populate it with the elements and attributes that make up listing 2. If we execute listing 3, by clicking on the button, the memo component displays the XML that is created - Figure 2 is a screenshot of the form after the code in listing 3 has been executed.

```
procedure TfrmXDOM.btnBuildXDOMClick(Sender:
                                TObject);
var
    doc: TdomDocument;
    node, nodel: TDomNode;
begin
    // Create a TdomDocument with richplum as
    // the root node
    doc:=
        xdomRichplum.createDocument('richplum',nil);

    // add the attribute display = "true" to
    // the document element
    doc.documentElement.setAttribute('display',
                                    'true');

    // create a node: <message></message>
    node := doc.createElement('message');

    // create a node: hello, world
    nodel:= doc.createTextNode('hello, world');

    // Build the element: <message>hello,
    // world</message>
    node.appendChild(nodel);

    // Append the node to the document tree
    doc.documentElement.appendChild(node);

    // Copy the raw XML into a memo
    memol.text:= doc.codeAsString;

    // Clear the document tree
    doc.clear;
end;
```

Listing 3 – XDOM using Delphi

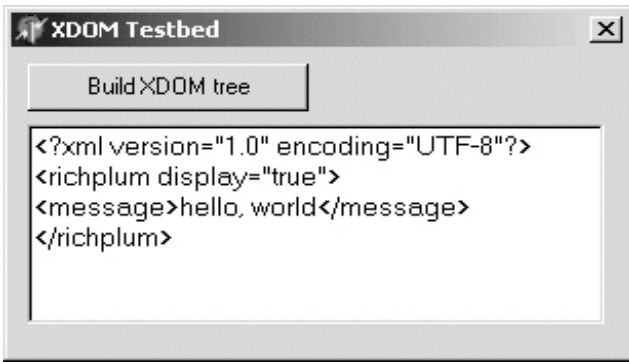


Figure 2

This is where it gets interesting – because we’re using an implementation of the DOM specification, the Delphi code we’ve just written is very similar (verging on portable) to code we would write in another language for another DOM implementation. Listing 4 presents a simple HTML page with some JavaScript that instantiates the Microsoft XML Parser and builds the same XML DOM. Figure 4 presents the XML created by the browser_dom() function, after the Build MSXML DOM button has been clicked.

```
<HTML>

<HEAD>

<SCRIPT LANGUAGE="JavaScript">
function browser_dom()
{
var doc = new
  ActiveXObject("Msxml2.DOMDocument");
var rootElement=doc.createElement("richplum");
doc.appendChild(rootElement);

// add the attribute display = "true" to
// the document element
doc.documentElement.setAttribute('display',
                                'true');

// create a node: <message></message>
var node=doc.createElement("message");

// create a node: hello, world
var node1=doc.createTextNode("hello, world");

// Build the element: <message>hello,
//                               world</message>
node.appendChild(node1);

// Append the node to the document tree
doc.documentElement.appendChild(node);

taXML.value = doc.xml;
}
</SCRIPT>

</HEAD>

<BODY>
<BUTTON ONCLICK="browser_dom()">BUILD MSXML
DOM</BUTTON>
<BR />
<TEXTAREA ID="taXML" ROWS="5" COLS="40" />
</BODY>

</HTML>
```

Listing 4 – MSXML DOM using JavaScript



Figure 3

If we compare listing 1 and listing 2, we can see a lot of overlap, however we can also see where each DOM implementation has been extended in some way. Most DOM implementations recognise that developers need a mechanism for extracting the XML that makes up an XML document. OpenXML extends the DOM by providing the codeAsString property. MSXML provides the same functionality via the XML property. How the DOM implementation allows for the instantiation of a DOM document is also an area where DOM extensions are prevalent. Object instantiation is frequently language-dependent, hence the need for extensions to the DOM implementation.

TdomNode is the key

23,000 lines of code might sound like a lot to learn in one go – however, once you have spent some time working with just one class, TdomNode, you’ll realise that it’s not as daunting as you first thought. Listing 5 presents the public properties and member functions that TdomNode has to offer. We look at some of these procedures and functions over the course of this article and the next.

```
TdomNode = class
public
  constructor create(const aOwner:
                                TdomDocument);
  destructor destroy; override;
  function appendChild(const newChild:
                                TdomNode): TdomNode; virtual;
  procedure clear; virtual;
  function cloneNode(const deep: boolean):
                                TdomNode; virtual;
  function hasChildNodes: boolean; virtual;
  function insertBefore(const newChild,
                                refChild: TdomNode): TdomNode; virtual;
  function isAncestor(const AncestorNode:
                                TdomNode): boolean; virtual;
  procedure normalize; virtual;
  function removeChild(const oldChild:
                                TdomNode): TdomNode; virtual;
  function replaceChild(const newChild,
                                oldChild: TdomNode): TdomNode; virtual;
  function resolveEntityReferences(const opt:
                                TdomEntityResolveOption): boolean;
                                virtual;
  function supports(const feature,
                                version: wideString): boolean; virtual;
  procedure writeCode(Stream: TStream);
                                virtual;
  property attributes: TdomNamedNodeMap
                                read getAttributes;
  property childNodes: TdomNodeList
                                read getChildNodes;
  property code: wideString
                                read getCode;
  property firstChild: TdomNode
                                read getFirstChild;
  property isReadOnly: boolean
                                read FIsReadOnly;
```

```

property lastChild: TdomNode
    read getLastChild;
property localName: wideString
    read FLocalName;
property namespaceURI: wideString
    read FNamespaceURI;
property nextSibling: TdomNode
    read getNextSibling;
property nodeName: wideString
    read getNodeName;
property nodeType: TdomNodeType
    read getNodeType;
property nodeValue: wideString
    read getNodeValue write setNodeValue;
property ownerDocument: TdomDocument
    read getDocument;
property parentNode: TdomNode
    read getParentNode;
property previousSibling: TdomNode
    read getPreviousSibling;
property prefix: wideString
    read FPrefix write setPrefix;
end;

```

Listing 5 – TdomNode class definition

The DOM provides live access to the XML elements that make up the in-memory DOM. Thus, any changes you make to a node, regardless of how you got hold of it, are automatically posted.

TdomDocument is also important

TdomNode is so important that even TdomDocument is a descendant of it. TdomDocument provides us with a number of member functions that allow the programmatic creation of an XDOM. Listing 6 presents the public methods TdomDocument has to offer. Again, I cover some of these methods in this article and the next.

```

TdomDocument = class (TdomNode)
public
    constructor create(const aOwner:
        TDomImplementation); virtual;
    destructor destroy; override;
    procedure clear; override;
    procedure clearInvalidNodeIterators;
        virtual;
    function createElement(const tagName:
        wideString): TdomElement; virtual;

    function createElementNS(const
        namespaceURI, qualifiedName: wideString):
        TdomElement; virtual;

    function createDocumentFragment:
        TdomDocumentFragment; virtual;
    function createTextNode(const data:
        wideString): TdomText; virtual;
    function createComment(const data:
        wideString): TdomComment; virtual;
    function createCDATASection(const data:
        wideString): TdomCDATASection; virtual;

    function createProcessingInstruction(const
        targ,data : wideString):
        TdomProcessingInstruction; virtual;

    function createAttribute(const name:
        wideString): TdomAttr; virtual;
    function createAttributeNS(const
        namespaceURI, qualifiedName: wideString):
        TdomAttr; virtual;
    function createEntityReference(const name:
        wideString): TdomEntityReference; virtual;

```

```

function createDocumentType(const name,
    pubId, sysId, intSubset: wideString):
    TdomDocumentType; virtual;

procedure freeAllNodes(var node: TdomNode);
    virtual;
procedure freeTreeWalker(var treeWalker:
    TdomTreeWalker); virtual;
function getElementById(const elementId:
    wideString): TdomElement; virtual;
function getElementsByTagName(const
    tagName: wideString): TdomNodeList;
    virtual;
function getElementsByTagNameNS(const
    namespaceURI,localName: wideString):
    TdomNodeList; virtual;
function importNode(const importedNode:
    TdomNode; const deep: boolean): TdomNode;
    virtual;

function insertBefore(const newChild,
    refChild: TdomNode): TdomNode; override;
function replaceChild(const newChild,
    oldChild: TdomNode): TdomNode; override;
function appendChild(const newChild:
    TdomNode): TdomNode; override;
function createNodeIterator(const root:
    TdomNode; whatToShow:

TdomWhatToShow;nodeFilter: TdomNodeFilter;
    entityReferenceExpansion: boolean):
    TdomNodeIterator; virtual;

function createTreeWalker(const root:
    TdomNode;whatToShow: TdomWhatToShow;
    nodeFilter: TdomNodeFilter;
    entityReferenceExpansion: boolean):
    TdomTreeWalker; virtual;

function removeContentModel: TdomCmObject;
    virtual;
function setContentModel(const arg:
    TdomCmObject): TdomCmObject; virtual;

function validate(const errorHandler:
    TdomCustomErrorHandler; const opt:
    TdomEntityResolveOption): boolean;
    virtual;

procedure writeCode(stream: TStream);
    override;
procedure writeCodeAsUTF8(stream: TStream);
    virtual;
procedure writeCodeAsUTF16(stream:
    TStream); virtual;
property codeAsString: string read
    getCodeAsString;
property codeAsWideString: wideString read
    getCodeAsWideString;
property contentModel: TdomCmObject read
    FCMInternal;
property defaultView: TdomAbstractView read
    FDefaultView;
property doctype: TdomDocumentType read
    getDoctype;
property documentElement: TdomElement read
    getDocumentElement;
property domImplementation:
    TdomImplementation read FDomImpl;
property encoding: wideString read
    FEncoding write FEncoding;
property IDs: TStringList read FIDs;
property standalone: wideString read
    FStandalone write FStandalone;
property systemId: wideString read
    FSystemId write FSystemId;
property version: wideString read FVersion
    write FVersion;
end;

```

Listing 6 – TdomDocument extends TdomNode

Working with OpenXML

The best way to understand the DOM and OpenXML is to work through a fairly simple example. The remainder of this article presents a simple example where we cover most of the features of the DOM and we see just how they're implemented in OpenXML.

Loading XML from a file

XDOM provides a mechanism that allows us to load XML from a file. For the remainder of this article we use listing 7 as reference – I've formatted listing 7 so that it looks neater in print, in reality you should remove the white space.

```
<?xml version="1.0"?>
<richplum>
<staff no="1"><surname>Goulson</surname>
<firstname>Phil</firstname></staff>
<staff no="2"><surname>Pooley</surname>
<firstname>Joanna</firstname></staff>
<staff no="3"><surname>Jack</surname>
<firstname>Angus</firstname></staff>
<staff no="4"><surname>Parsons-Hann</surname>
<firstname>Wendy</firstname></staff>
<staff no="5"><surname>Jenkinson</surname>
<firstname>Debra</firstname></staff>
<staff no="6"><surname>Jenkinson</surname>
<firstname>Jon</firstname></staff>
<staff no="7"><surname>Wintringham</surname>
<firstname>Ben</firstname></staff>
<staff no="8"><surname>Scott</surname>
<firstname>Steve</firstname></staff>
</richplum>
```

Listing 7 – richplum.xml

Listing 8 presents the few lines of code required to demonstrate loading XML from a file. Actually, there's only one line required – the other lines are purely peripheral. The fileToDom method allows our Delphi/Kylix application to load XML that has previously been created – perhaps by another application. In a typical business-2-business scenario it is likely that another application has created an XML document for our application to consume/process. This is perhaps the most motivating factor – XML is good at representing data in a non-proprietary format. You might argue that traditional Comma Separated Value (CSV) files are just as good. Well, XML also allows us to include metadata – something that is not impossible using CSV files, but something that is certainly tricky to implement. So, typically we can expect to see XML being used to transfer data between applications – this is an area that is becoming much more prevalent in today's web-centric applications.

```
procedure TForm1.btnLoadXMLClick(Sender:
TObject);
begin
// Load the pre-prepared XML file into an
// XDOM...
doc:=XmlToDomParser.fileToDom('richplum.xml');

memo1.text := doc.codeAsString;

btnPopulate.Enabled:=true;
end;
```

Listing 8 – loading XML from a file

Iterating over nodes

Moving from node to node is achieved using the childNodes (TdomNodeList) collection or via the previousSibling, nextSibling, firstChild and lastChild member functions. The hasChildNodes member function is [obviously] useful in determining whether a node has child nodes – typically we would use this function in our own recursive function. We'll look at advanced node traversal next time.

Listing 9 presents some sample code that obtains a collection of the <staff> elements from Listing 7. The collection of <staff> element is then traversed using a *for* loop. Listing 10 presents the same functionality using a *while* loop. Whichever method you prefer, Figure 4 presents our results: we've loaded a previously created XML file into an XDOM, then iterated over the elements presenting them in a TListView instance.

```
procedure TForm1.btnPopulateClick(Sender:
TObject);
var xmlStaffList      : TDomNodeList;
    xmlNodeMap        : TDomNamedNodeMap;
    xmlSurnameNode    ,
    xmlFirstNameNode  : TDomNode;

    listItem          : TListItem;
    nStaffNo          : Integer;
    i                  : Integer;
begin
// 'Get' all the <STAFF> elements...
xmlStaffList :=
    doc.getElementsByTagName('staff');

for i:=0 to xmlStaffList.Length - 1 do begin
// Obtain access to the attributes for
// this element...
xmlNodeMap :=
    xmlStaffList.item(i).attributes;

sStaffNo :=
    xmlNodeMap.GetNamedItem('no').NodeValue;
xmlSurnameNode :=
    xmlStaffList.item(i).childNodes.item(0);
// OR xmlSurnameNode :=
//    xmlStaffList.item(i).firstChild;

xmlFirstNameNode :=
    xmlStaffList.item(i).childNodes.item(1);

// Add the staff member to the ListView...
listItem := ListView1.Items.add;
listItem.Caption := sStaffNo;

listItem.SubItems.Add(
    xmlSurnameNode.firstChild.nodeValue);
listItem.SubItems.Add(
    xmlFirstNameNode.firstChild.nodeValue);
end;
end;
```

Listing 9 – iterating over XML nodes

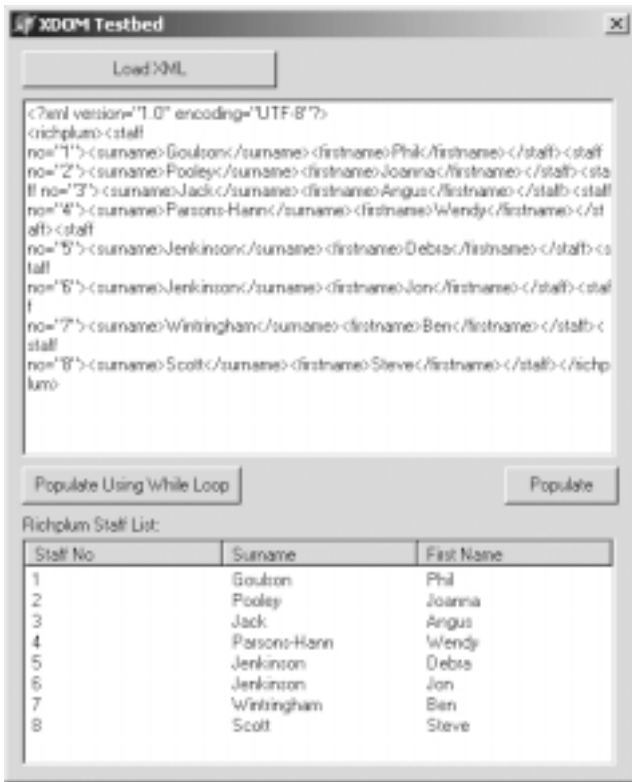


Figure 4

```

procedure
TForm1.btnPopulateUsingWhileClick(Sender:
      TObject);
var xmlStaffList      : TDomNodeList;
    xmlNodeMap        : TDomNamedNodeMap;
    xmlSurnameNode,
    node, xmlFirstNameNode : TDomNode;

    listItem         : TListItem;
    sStaffNo         : String;
    i                 : Integer;

    nt : TdomNodeType;
begin
// 'Get' all the <STAFF> elements...
xmlStaffList :=
    doc.getElementsByTagName('staff');

node := xmlStaffList.item(0);
while node <> nil do begin
    xmlNodeMap := node.attributes;

    sStaffNo :=
        xmlNodeMap.GetNamedItem('no').NodeValue;

    xmlSurnameNode :=node.childNodes.item(0);

    xmlFirstNameNode :=node.childNodes.item(1);

// Add the staff member to the ListView...
    listItem := Listview1.Items.add;
    listItem.Caption := sStaffNo;
    listItem.SubItems.Add(
        xmlSurnameNode.firstChild.nodeValue);
    listItem.SubItems.Add(
        xmlFirstNameNode.firstChild.nodeValue);

    node:=node.nextSibling;
end;
end;

```

Listing 10

Listing 10 works because of the data structure behind the DOM. Given a single node, the DOM has interfaces that permit the navigation back to the previous node and forward to the next node – the OpenXML XDOM implements those interfaces. Similarly, the same single node has pointers to a collection (NodeList) of child nodes – we can iterate over child nodes using the collection or using *firstChild*, *nextSibling* navigation. Each node also has a collection of attributes – these are accessible via a NamedNodeMap. Attributes are represented as DomNodes; thus everything in the DOM and XDOM is accessible through the TDomNode object. Figure 5 depicts this structure graphically.

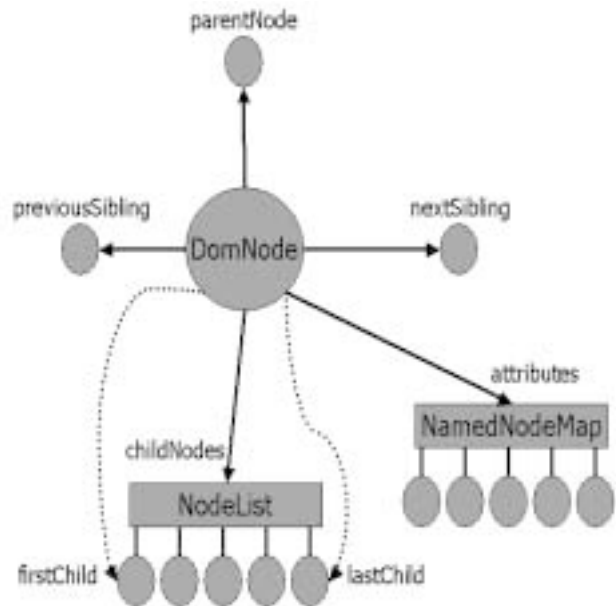


Figure 5

TdomDocument allows querying (of sorts)

Listing 6 provides us with two member functions that permit structured access to our XDOM elements – when I say structured, I don't mean structured as in SQL. The functions *getElementsByTagName* and *getElementsByTagNameNS* allow us to extract all the elements that have the same name or namespace. We see an example of *getElementsByTagName* later in this article. Next time, we look at namespaces and more advanced filtering of elements.

Given the following snippet, we are able to extract all the <staff> elements into an instance of TDomNodeList:

```

var xmlStaffList      : TDomNodeList;
...
// 'Get' all the <STAFF> elements...
xmlStaffList :=
    doc.getElementsByTagName('staff');

```

I must stress that xmlStaffList is a live view into your XDOM – if you make any changes to the node values they are updated automatically. This is perhaps the one area that might catch you out – after all, we're all used to working with [non-live] snapshots of our data.

There's more to querying/filtering than I have shown here – we go into more detail in part two of this article.

Summary

OpenXML allows you to incorporate XML support into your applications – there is no additional baggage, runtimes, or DLLs to worry about. OpenXML implements what is intended to be a portable Internet standard – thus the time that you invest in learning OpenXML will give you a skill that you can use not just in an HTML page, but on non-Intel platforms too.

Rightly or wrongly, XML is being used everywhere – developers want to incorporate XML into their applications, managers are using XML to win clients and product vendors/marketing departments are desperate to show off their latest products with native XML support or the like.

By using XML, whether it is the OpenXML implementation or another vendor's, we are able to provide a structured mechanism for information interchange between other applications and ultimately other businesses. Thus XML is good for business and it is good for business-2-business – introducing XML into your business ultimately introduces profit.



Craig works as an Enterprise Developer (and Dilbert Evangelist!) for Currie & Brown (<http://www.currieb.com>) – their primary business is quantity surveying, cost management and project management. He can be reached via e-mail at: Craig.Murphy@currieb.co.uk or

Craig@isleofjura.demon.co.uk

OpenXML - Resources

[1] XML Schema Part 2: Data Types:
<http://www.w3.org/TR/xmlschema-2/>

[2] OpenXML is available from this web site:
<http://www.philo.de/xml> .

The OpenXML manual in HTML Help (.chm) format:
http://www.philo.de/xml/dom/xdom2_3_10.chm.
It's worth noting that at the time of writing this .chm file documented an earlier version of the XDOM source code. Snippets of the OpenXML source are presented in this article - The author of OpenXML, Dieter Köhler, gave his permission to use code fragments in this article.

[3] Charlie Calvert has written two articles about OpenXML. Basic XML Parsing in Delphi:
<http://homepages.borland.com/ccalvert/TechPapers/Delphi/XMLSimple/XMLSimple.html>

Intermediate Level XML Parsing with Delphi: <http://homepages.borland.com/ccalvert/TechPapers/Delphi/XMLBrowse/index.htm>

[4] The W3C DOM Level 2 specification is available here: <http://www.w3c.org/DOM-Level-2/>

As usual, all the source code for this article is available from my web site:
<http://www.isleofjura.demon.co.uk/bug>.

Software Developers vs Dru

Drug Dealers	Software D
Refer to their clients as "users"	Refer to their clie
"The first one's free!"	"Download a free
Have important Asian connections	Have important A
Strange jargon: "Stick" "Rock" "Wrap" "E" "Stash" "Drive-by" "Hit (LSD)" "Source" "The Pigs"	Strange jargon: "SCSI" "RTFM" "Packet" "C" "Cache" "CTRL ALT DE" "Hit (WWW)" "Source-code" "Microsoft"
Realise that there's tons of cash in the 14- to 25-year-old market	Realise that there old market
Clients really like your your stuff when it works. When it doesn't work they want to kill you	Clients really lik it doesn't work th
Job is assisted by the industry's producing newer, more potent product	Job is assisted by potent product
Often seen in the company of pimps, hustlers and low-lifes	Often seen in the venture capitalist
When things go wrong, a "fix" is just a phone call away, but may be expensive	When things go w away, but may be
A lot of people are getting rich while still teenagers	A lot of people a
Product causes unhealthy addictions	DOOM, Quake,
Do your job well and you can sleep with sexy movie stars who depend on you	Damn! DAMN!!!