Introducing the Document Object Model using OpenXML (Part II)

by Craig Murphy

In my last article I provided an introduction to the OpenXML project, a project that encompasses the eXtended Document Object Model (XDOM) XML components. Since then Borland has shipped Delphi 6 – it seems fair to provide a brief comparison/commentary on the XML features in Delphi 6. You only have to glance at the Delphi 6 features matrix to see that Borland have provided considerable support for XML in the form of DataSnap and BizSnap. However, DataSnap and BizSnap are features of Delphi 6 Enterprise thus, if you are using Delphi 6 Personal or Professional, you may find XML support is limited to MyBase (Personal XML Database Engine).

The XDOM implements the W3C DOM Core Level 2 recommendation [1] – the same standard that Borland have provided in Delphi 6. However, as I mentioned in part 1, the DOM is a portable Internet standard, therefore the time you spend learning the DOM interfaces is time well spent – you will be able to make use of any DOM on any platform. The Delphi 6 help confirms this:

"At the lowest level in the XML document programming support is the new xmldom.pas interface unit that provides a cross platform and vendor independent set of interfaces for programming with the W3C DOM Level 2 specification. Designed with an open architecture, the interfaces are easily integrated with existing DOM-based XML solutions."

Delphi 6 provides a set of interfaces to third party DOM implementations such as the Microsoft XML parser (MSXML) or the IBM XML Parser. You will need to be confident that your client machines have the third party parser installed. This is where OpenXML has an edge – it is a completely standalone solution that does not rely on any additional DLLs. Using OpenXML does not necessitate the installation of Internet Explorer 5.x on your client machines – thus your install scripts are kept free from that complexity too.

Recap

In part one of this article, I covered enough of the OpenXML components to allow you to start using XML in your applications. We looked at how the XDOM extended the W3C DOM to allow XML files to be loaded. We also examined how we could programmatically create an XML document, and how we would select certain elements within an XML document. The W3C DOM was introduced and positioned as a portable Internet standard – so portable, we were able to demonstrate building a DOM in Delphi and in Internet Explorer 5. Importantly we saw that, regardless of how you came about an XML document, it is a live view into the XDOM data.

Where are we going?

In part one, I promised to cover node types in more detail. I also plan to cover TdomNode and the descendant class TdomDocument in more detail. Namespaces were skimmed over, so I will explain them too. I am going to assume that you have read part one of this article. When I wrote it, XDOM was at version 2.3.10 and consisted of about 23,000 lines of code. Now, July 2001, XDOM is at version 2.3.15 and consists of over 29,000 lines of code. If you plan to use the XDOM components, it's well worth visiting the OpenXML[2] web site.

Listing 1 presents the XML document that will be used throughout this article; most of the code snippets will also assume that listing 2 has been executed beforehand.

TdomNode

TdomNode is a central XDOM class – we will be using many of the TdomNode functions and properties over the course of this article. Listing 3 presents the class definition for TdomNode. Iterating over an XML document using the nextSibling and previousSibling properties was demonstrated in part one – this time we will take a look at what else TdomNode has to offer.

Given that XML is good at representing hierarchical information, surely there must be a means of identifying whether a given node has child nodes? Well, there is – the hasChildNodes method does just that. This function is particularly good for use in a recursive function that iterates over nodes - we will see an example of hasChildNodes later in this article.

DOM Node Types

If we examine listing 3, we can see that every node has a nodeType property. nodeType is of type TdomNodeType – Listing 4 presents the node types as implemented in XDOM.pas. In part one I explained that a TdomNode could represent everything in an XML document and that includes the elements and attributes that we have seen so far. Well, it gets better, even the comments and textual aspects of an XML document can be represented using a TdomNode. Listing 5 presents a recursive routine that traverses an entire XML document, picking out the nodeTypes on the way.

The code in listing 5 is not exactly rocket science, however it does demonstrate the principle behind iterating over an XML document of unknown size – with a little modification, it could easily be used to populate a treeview.

The example application that accompanies this article has a "Node Types" tab – assuming that listing 1 has been loaded, and listing 5 has been executed, Figure 1 presents the richplum.xml node types.

Selective Traversal

In part one I mentioned that we would be looking at some more advanced filtering techniques. True to my word, I will now discuss how the XDOM allows us to filter an XML document at an element-level. The XDOM implements the W3C Document Object Model Traversal [3] interfaces: Iterator and TreeWalker. Both of these interfaces use NodeFilters to implement the notion of document mutation. A NodeFilter is essentially a user-provided function that "accepts", "rejects" or "skips" a node.

Using an Iterator

So, referring to listing 1, imagine how we might effect the following filter: *list all <staff> elements and their children, where the <staff> 'no' attribute is **not** divisible by two.* It's a trivial filter, but it does provide enough attributes [no pun intended] to allow me to demonstrate XDOM filters. Whilst we could easily write our own iteration and selection routine, we would find ourselves writing the same piece of code over and over again. The XDOM allows a generic filter event to be fired each and every time we iterate over a given XML document.

To implement a generic filter we must first build a class that descends from TdomNodeFilter. Listing 6 presents the snippet of code required to define our own TrpIteratorFilter class. Listing 7 implements the TrpIteratorFilter class – there is only one method: acceptNode. Now that we have a mechanism that allows filtering, we must connect it to a TdomNode. If you recall from part one, a TdomNode knows about the nodes that come before and after it, so a single node may well allow us to navigate through a collection of nodes. XDOM offers a TdomNodeIterator class that provides us with a means of attaching the filter event, acceptNode, to a given TdomNode – thus to iterate over an entire DOM document, we can create an instance of TdomNodeIterator for the documentElement.

Listing 8 presents TdomNodeIterator's create method definition – it takes four parameters, the first three of which are useful in our example. The root parameter specifies the element that we are going to iterate over, in our case it will be doc.documentElement, i.e. <richplum>. The second parameter, whatToShow, takes values from those specified in listing 4 – notice how we are reusing the TdomNodeType mentioned earlier. The third parameter takes an instance of a TdomNodeFilter. Listing 9 presents a snippet of the XDOM source – specifically the types and constants used by Iterators.

The whatToShow parameter may cause a little confusion – it is capable of providing a first-line filter. In other words, by the time the NodeFilter function is executed, we will already by looking at a subset of the original XML document. Thus because I have set whatToShow equal to nt_ElementNode, any node type that is not an element is not considered to be part of the view. The W3C refer to this side effect as node visibility.

Listing 10 presents the code required to implement an Iterator. Figure 2 presents a screenshot from an application that puts the code in listings 6, 7 and 10 to good use.

The power of the TdomNodeFilter class can be harnessed if we consider how easy it is to extend our descendant class to provide some serious customisation via the addition of additional properties. The acceptNode function simply alters the filter it performs based on it's state – no longer do we need to cut and paste code that iterates over an XML document selectively filtering out some elements. Essentially, filters of this nature provide us with a means of programmatically implementing [SQL] views.

The acceptNode function may return one of three values: filter_accept, filter_reject or filter_skip. filter_accept means that the node is acceptable and will form part of the logical view. filter_reject is also fairly obvious, the node and any children will not form part of the logical view. filter_skip is slightly different – it means that the current node is not part of the logical view, but any children it has "may be" part of the view.

Before we move on, it's worth noting that TdomNodeIterator is slightly different from the other DOM traversal classes – it offers forward and reverse traversal via the nextNode and previousNode methods, there are no nextSibling, previousSibling methods.

Using a TreeWalker

On first impressions, you could easily think that a TreeWalker and an Iterator performed the same task. In essence they do, the main difference being the data structure each has to offer. As we have just seen, the Iterator offers a linear forward/reverse list-like view into the DOM. The TreeWalker, on the other hand, offers a three-dimensional view that allows us to examine the children and the parent of the currently selected node.

Listing 11 presents an example of the TreeWalker in use – again it's a trivial example, however it demonstrates that we can traverse across (nextNode) the node tree as well as up (parentNode). Figure 3 presents a screenshot of the TreeWalker in action.

A word of warning…

We have seen the power offered by NodeFilters. Use that power wisely – the W3C DOM Traversal Recommendation advises against using NodeFilters as a mechanism for modifying the DOM tree. However, NodeFilters do react well to an ever-changing XML document – you may add and remove elements from an XML document without affecting any associated Iterators or TreeWalkers.

XML Namespaces

XML Namespaces provide a means of preventing element naming conflicts and allow us to "infer" some sort of meaning from XML elements (and XML element groups). XDOM provides a number of functions that help us work with XML namespaces. Listing 12 presents an XML document that has been augmented with namespaces. The crux behind Listing 12 is the uniqueness that we can derive from a web address (a URL). Notice how we have two invoice elements, each prefixed with a namespace: richplum and poorpeach. Each namespace is associated with an identifier – this identifier really has to be unique, hence we use [permanent] URLs. Any elements inside a prefixed element are automatically assumed to be members of the same namespace, i.e. <amount> is a member of the richplum namespace.

The XDOM provides a considerable number of 'helper functions' that allow us to work with raw XML. Listing 13 presents a few of these functions. Given <richplum:invoice>, we can use XMLExtractPrefix to obtain 'richplum' or 'poorpeach'; whereas XMLExtractLocalName allows us to obtain the invoice component. Using isXMLName allows us to determine whether richplum:invoice is a valid XML name.

Summary

Over the course of these two articles, we have seen how to incorporate XML into our day-to-day applications. The importance of W3C recommendations cannot be understated. W3C recommendations are typically portable Internet standards, thus learning about them and using them should always be time well spent. Other W3C recommendations that are in the W3C space include the Simple Object Access Protocol (SOAP) and the eXtensible Stylesheet Language for Transformation (XSLT).

Even with the recent release of Delphi 6, there is still a market for OpenXML – not everybody will upgrade to Delphi 6 immediately. Equally, Delphi 6 Enterprise may well be financially prohibitive. OpenXML is currently free – it works with Delphi 3,4,5 and Delphi 6 (at the time of writing this involved minor tweaks to the uses clause – no doubt the author is working on a version that works in all versions, even Kylix).

*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*

Resources

[1] The W3C DOM Level 2 specification is available here: http://www.w3c.org/DOM-Level-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_15.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] The W3C DOM Traversal recommendation is available here: http://www.w3.org/TR/DOM-Level-2-Traversal-Range

All the source code for this article is available from my web site: http://www.craigmurphy.com/bug.

```
<?xml version="1.0"?>
<!– richplum.xml –>
<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 1 – richplum.xml

```
doc : TDomDocument;
doc:=XmlToDomParser.fileToDom('richplum.xml');
```

Listing 2 – Assumed state

Figure 1 – Every node has a type

Figure 2 – Advanced filtering really does work

Figure 3 – TreeWalker allow access to parent nodes (and children)

```
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 3 – TdomNode class definition

```
  TdomNodeType = (ntUnknown,
           ntElement_Node,
           ntAttribute_Node,
           ntText_Node,
           ntCDATA_Section_Node,
           ntEntity_Reference_Node,
           ntEntity_Node,
           ntProcessing_Instruction_Node,
           ntComment_Node,
           ntDocument_Node,
           ntDocument_Type_Node,
           ntDocument_Fragment_Node,
           ntNotation_Node);
```

Listing 4 - TdomNodeType

```
procedure
TForm1.btnParseNodeTypeClick(
           Sender: TObject);
var
   xmlNode : TDomNode;
function _TypeOf(n : TdomNode) : string;
   var
     sType : string;
   begin
     case n.nodeType of
           ntAttribute_Node: sType :=
             'ntAttribute_Node';
           ntElement_Node:   sType :=
             'ntElement_Node';
           ntComment_Node:   sType :=
             'ntComment_Node';
           ntText_Node:      sType :=
             'ntText_Node';
           // ...abbreviated...
         end;
         _TypeOf := sType;
      end;

procedure _Recurse(n : TdomNode);
begin
   while n<>nil do begin
     Listbox1.items.add(n.nodeName +
           '('+ n.nodeValue + ') is a
     ' + _TypeOf(n));
           if n.hasChildNodes then
            _Recurse(n.firstChild);
            n:=n.nextSibling;
          end;
   end;

begin
    Listbox1.Clear;
    xmlNode := doc.firstChild;
    while xmlNode <> nil do begin
     Listbox1.items.add(xmlNode.nodeName +
       '('+ xmlNode.nodeValue + ') is a
       ' + _TypeOf(xmlNode));
    if xmlNode.hasChildNodes then
      _Recurse(xmlNode.firstChild);
      xmlNode:=xmlNode.nextSibling;
      end;
    end;
```

Listing 5 - Recursively examining nodes

```
TrpIteratorFilter = Class(TdomNodeFilter)
function acceptNode(const n: TdomNode):
    TdomFilterResult; override;
end;
```

Listing 6 – Filtering is inherited from TdomNodeFilter

```
function TrpIteratorFilter.acceptNode(
    const n: TdomNode): TdomFilterResult;
var  xmlNodeMap : TDomNamedNodeMap;
     sStaffNo   : string;
begin
  if n.nodeName = 'staff' then begin
     xmlNodeMap := n.attributes;
     sStaffNo  := xmlNodeMap.GetNamedItem(
           'no').NodeValue;
     if (StrToInt(sStaffNo) mod 2) = 0 then
          result := filter_reject
     else
          result := filter_accept;
     end
  else
     result:=filter_reject;
end;
```

Listing 7 – implementing advanced filtering

```
constructor create(const root: TdomNode;
      const whatToShow: TdomWhatToShow;
      const nodeFilter: TdomNodeFilter;
      const entityReferenceExpansion:
      boolean); virtual;
```

Listing 8 – TdomNodeIterator's create method is the key

```
Type TdomWhatToShow = set of TdomNodeType;

TdomFilterResult =
 (filter_accept,filter_reject,filter_skip);

const show_all: TdomWhatToShow =
 [ntElement_Node .. High(TDomNodeType)];
```

Listing 9 – Filling in the blanks

```
procedure TForm1.btnIteratorClick(Sender
          : TObject);
var domItr  : TdomNodeIterator;
    filter  : TrpIteratorFilter;
    node    : TdomNode;

begin
    // Create node filter
    filter := TrpIteratorFilter.Create;
    try
    // Create node iterator
    domItr:=TdomNodeIterator.create(
          doc.documentElement,
         [ntElement_Node],filter,false);
    memo2.clear;
    node := domItr.nextNode;
    while node <> nil do begin
          memo2.text := memo2.text +
                node.code + #13#10;
         node:=domItr.nextNode;
         end;
    finally
       filter.free;
    end;
end;
```

Listing 10 – Using an Iterator

```
procedure TForm1.btnTreeWalkerClick(
          Sender: TObject);
var domTW   : TdomTreeWalker;
    filter  : TrpTreeWalkerFilter;
    node    : TdomNode;

begin
    // Create node filter
    filter := TrpTreeWalkerFilter.Create;
    try
      // Create node iterator
     domTW:=TdomTreeWalker.create(
         doc.documentElement,
        [ntElement_Node], filter, false);
     memo4.clear;
     node := domTW.nextNode;
     while node <> nil do begin
      memo4.text := memo4.text + node.code;
     if (node.parentNode <> nil) then
     memo4.text := memo4.text + ' my parent
           is ' + node.parentNode.nodeName;
     memo4.text := memo4.text + #13#10;
     node:=domTW.nextNode;
     end;
  finally
   filter.Free;
  end;
end;
```

Listing 11 – Using a TreeWalker

```
<?xml version="1.0"?>
<root>
<richplum:invoice xmlns:richplum=
    "http://richplum.co.uk/invoice">
<amount>100</amount>
</richplum:invoice>

<poorpeach:invoice xmlns:poorpeach=
    "http://poorpeach.co.uk/invoice">
<amount>100</amount>
</poorpeach:invoice>
</root>
```

Listing 12 – namespaces in an XML document

```
function isXmlName(const S: wideString):
    boolean;
function XMLExtractLocalName(
    const qualifiedName: wideString):
    wideString;
function XMLExtractPrefix(
    const qualifiedName: wideString):
    wideString;
```

Listing 13 – XDOM namespace helper functions