Introduction

More and more Xbase++ developers are needing to deal with XML in their applications. As systems go paperless, the electronic transfer of information becomes more and more relevant. This transfer of information is most often in the form of XML.

This session discusses concepts for dealing with XML both from the need to parse XML data into existing applications to the need to generate XML to send to other applications.

Attendees will receive source code for an XmlNode class that has methods for reading, writing, managing and creating XML streams.

* The basics of XML format.

* Why do we need XML? How does it compare to HTML?

* How to use the Xbase++ XML parser.

* What is the difference between callback parsing and heirarchal parsing?

* How to use the XmlNode class.

* How to convert DBF data to XML and vice-versa.

* How to convert objects and arrays to XML and vice-versa.

* How to work with an XML tree as a set of objects.

* Embedding binary documents into XML with Base64 encoding.

* Embedding other XML or HTML documents into XML with CDATA or Escaping characters.

* Understanding Namespaces.

* Understanding SOAP and WSDL.

* New XML functions in Xbase++ 2.0.

More and more Xbase++ developers are needing to deal with XML in their applications. As systems go paperless, the electronic transfer of information becomes more and more relevant. This transfer of information is most often in the form of XML.

This session discusses concepts for dealing with XML both from the need to parse XML data into existing applications to the need to generate XML to send to other applications.

The Basics of XML format

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined by the W3C's XML 1.0 Specification.

The design goals of XML emphasize simplicity, generality and usability across the Internet. It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures such as those used in web services.

XML enables you to define a grammar for marking up documents in terms of semantic tags and their attributes. This is a quite good definition of XML. However, don't be disappointed if it doesn't tell much to you. There are some terms that need an explanation:

Grammar - This term describes the words allowed in a language and how or where they may appear in a sentence.

Semantic - This term describes the meaning of the words allowed in a language. E.g. "Deutsch" is not part of the English language, it is part of the German language. The semantic of the words "Deutsch" and "German" is identical, depending on the language.

Mark-up - This term describes all characters in a text that define how text is displayed or printed. For example, italic or bold markers are included in a text but do not contribute to its contents. The markers are invisible when the text is displayed or printed. However, they are visible in an ASCII or HEX editor.

Tag - A tag is a word enclosed in an opening and closing delimiter. <HTML> is a tag existing in all HTML documents, for example.

Attribute - An attribute is part of a tag and defines additional information for mark-up. For example, <TR ALIGN="RIGHT"> is a string defining the HTML tag <TR> whose ALIGN attribute has the value "RIGHT".

With these "definition of terms", the XML definition above is much better to understand: XML allows you to define your own grammar for mark-up, i.e. in XML you can define the names for tags and their meaning, the context in which they may appear, and you can define attributes for tags and their meaning. As a matter of fact, XML is a meta-language that allows you to define your own language to describe documents, or data structures, and that is what makes XML the file format of choice for managing configuration data.

Since XML is a mark-up language, similar to HTML, there are some rules to comply with. The first rule is that all XML files must be "well-formed". This means that textual content in a file must be embedded in an opening and closing tag. The second rule requires all XML files to begin with an <?xml?> tag which identifies the file as being an XML file and includes version information. This is the only tag that has no closing pendant. The following XML code illustrates a simple, well-formed XML document used to describe the data structure for a customer address:

```
<?xml version="1.0"?>
<Customer>
  <Address>
```

```
    <Street>1562 FIRST AVE</Street>
    <City>New York City</City>
    <Zip>KOC 1HO</Zip>
    <State>NY</State>
    <Country>USA</Country>
  </Address>
</Customer>
```

This example demonstrates how data is organized in XML files. They contain information that can be represented as a tree, or as a parent/child hierarchy. <Customer> has a child tag <Address>, <Address> has child tags <Street>, <City>, <Zip> and so forth. The opening tag defines the meaning (semantic) of embedded information and the closing tag defines the end of data. The tag names define the allowed words (grammar) for a data structure.

As you can see, XML is a pretty simple file format. It is flexible enough to manage data of high complexity that can be stored in an ASCII compliant file format. XML files can be used anytime you have to store information outside a database, or when you have to exchange information between applications. As a matter of fact, whenever you need a file format to exchange data or to store configuration data - simply start drafting your own XML document. You are always on the safe side with XML.

Furthermore, because XML is already a well accepted standard, you can expect more and more tools to become available over time which ease the editing, parsing and data-exchange of information based on XML. There are a couple of tools available already to edit XML documents - one is Microsoft's XML Notepad (http://msdn.microsoft.com/xml/notepad/intro.asp), another more sophisticated one is XMLSPY (http://www.xmlspy.com).

Why Do We Need XML? How does it compare to HTML?

XML is a language that's very similar to HTML. It's much more flexible than HTML because it allows you to create your own custom tags. However, it's important to realize that XML is not just a language. XML is a meta-language: a language that allows us to create or define other languages. For example,with XML we can create other languages, such as SOAP, WSDL or even HTML.

We need XML because HTML is specifically designed to describe documents for display in a Web browser, and not much else. It becomes cumbersome if you want to display documents in a mobile device or do anything that's even slightly complicated, such as translating the content from German to English. HTML's sole purpose is to allow anyone to quickly create Web documents that can be shared with other people. XML, on the other hand, isn't just a web language. It can be used in a variety of different contexts, some of which may not have anything to do with humans interacting with content, for example, Web Services use XML to send requests and responses back and forth or storage of data in a heirarchal format.

HTML rarely (if ever) provides information about how the document is structured or what it means. Basically, HTML is a presentation language, whereas XML is a data-description language.

For example, if you were to go to any ecommerce Website and download a product listing, you'd probably get something like this:

```
<html>
 <head>
   <title>Alaska Software Products</title>
 </head>
```

```
   <body>
     <h1>Alaska Software Products</h1>
     <h2>Xbase++ 2.0 Foundation</h2>
     <p>The Basic version of Xbase++ 2.0</p>
     <p><b>Cost: $1500.00</b></p>

     <h2>Xbase++ 2.0 Professional</h2>
     <p>The Professional version of Xbase++ 2.0</p>
     <p><b>Cost: $2500.00</b></p>
   </body>
</html>
```

HTML tags such as <h1>, <h2> and <p> describe the presentation of the data including font and color. The are not case sensitive and web browser's parsers tend to be very "forgiving" of incorrect syntax, such as a missing closing tag.

```
<?xml version="1.0"?>
<productListing title="Alaska Software Products">
  <product>
    <name>Xbase++ 2.0 Foundation</name>
    <description>The Basic version of Xbase++ 2.0</description>
    <cost>$1500.00</cost>
  </product>
  <product>
    <name>Xbase++ 2.0 Professional</name>
    <description>The Professional version of Xbase++ 2.0</description>
    <cost>$2500.00</cost>
  </product>
</productListing>
```

XML tags are case sensitive and the XML must be properly structured, otherwise XML parsers will return an error when reading the XML. In the above example, there is nothing in the data that implies any kind of visual style. It only represents data which can be parsed and utilized by a computer program.

There are other associative languages such as DTD and XSL which can be used with XML for validation rules or style rules, however this session will concentrate only on XML that is formatted for transfer of data.

How to use the Xbase++ XML Parser

The Alaska Software XML (ASXML) implementation provides an extremly fast lightweight XML processor. To give you an idea of its performance: it can process approximately 250.000 XML tags per second on a Dell Inspiron Laptop with a 433 MHz CPU.

The major design goals for the XML processor have been: simple usage, speed and a callback processing architecture. Callback processing simplifies the development of XML readers tremendously. Unlike Microsoft's XML implementation, where the parser creates a complete tree structure representing an entire XML document, a callback parser allows you to register own functions to process specific tags in a specific hierarchy.

For example, with a hierarchical parser you have to "walk through" the tree structure until you reach the node/tag of interest. The following pseudo code shows you the steps necessary if you would want to extract the contents of the <load> tag from an XML file using a hierarchical parser:

```
oRoot   := getRootTag()
oParent := oRoot:getChild( "DatabaseEngines" )
aChild  := oParent:getChildren( "load" )
FOR i:=1 TO Len( aChild )
  oChild := aChilds[i]
  // do whatever you want to to with the <load> tag
```

```
NEXT i
```

With a hierarchical parser you would extract the start, or root, tag and iterate through the tree until the desired tag is reached. In contrast, a callback processing parser uses a different approach. It assumes the user to register a function for a specific group of tags, or nodes, and passes the tags of interest along with associated data to that function. This means, the user associates an action with a tag and the parser starts traversing the tree and would execute all callback functions registered for specific tags. The following example demonstrates this technique and shows how easy it is to process the <load> tag using a callback processing approach.

```
registerFunction( "//DatabaseEngines/load", "myFunction" )
processDocument()

FUNCTION myFunction(oTag)
// do whatever you want to do with the <load> tag
RETURN(.T.)
```

The program code for the callback processing approach is not only much easier to read, it is also easier to maintain if the XML tag definition changes. We just have to register a function to be called for a specific node in the XML document structure ("//DatabaseEngines/load") and then we implement this the callback function to process the tag we are interested in. If the structure of a XML configuration file changes, there is no need to adapt PRG source code to reflect the changes, because the code is executed on a "per tag" basis and is independent of the physical structure of the XML document. The function ProcessConfig1() sample below shows this method of parsing.

Introduction to Heirarchal Parsing.

Callback parsing has its advantages as described above, but there are also many advantages to heirarchal parsing. Look at the sample program named DbeLoad.Prg. This sample demonstrates the reading of an XML file that is used as a configuration for loading DBEs in an application. The callback technique actually uses twice as many lines of code in this particular sample than the heirarchal technique. Also, the callback technique requires that the <load> tags in the XML are positioned above the <build> tags to insure that the dbes are loaded first, otherwise a runtime error will occur. The heirarchal technique does not have this requirement.

The source file named DCXML.PRG contains the source for a function and a class That uses the Xbase++ parser to create an XML DOM (Document Object Model) which is a heirarchal tree of the entire XML document as a set of node objects.

DC_Xml2ObjectTree() creates a DC_XmlNode() tree by parsing out an XML file or an XML stream. The DC_XmlNode() class contains methods for finding a child node, set of child nodes, reading attributes and content, and also rendering the object tree as text, array or XML stream. DC_XmlNode() can be used to create child nodes or a complete XML DOM for generating properly formatted XML. The function ProcessConfig2() sample below shows this method of parsing.

The Xbase++ XmlConfigLoad() / XMLConfigSave() classes

Xbase++ contains 2 classes which are specially designed for loading and saving configuration data that is formatted as XML. The idea of these two helper function is to simplify configuration management in applications. Fo that an XML format was specified with the following extended capabilities:

- XML elements as well as attributes can be accessed like a member var.
- Child nodes with the same name get automatically collected as an array of values at the parent node (implicit folding).
- XML files can link (include) other xml files transparently.
- XML comments are ignored/supported (of course).
- XML namespace is supported.
- Optional fixed type of nodes using valtypes, be default any attribute is considered a string.

The function ProcessConfig3() sample below shows this method of parsing.

```
//DbeLoad.Xml

<?xml version="1.0"?>
<Configuration>
  <DatabaseEngines>
    <load>DBFDBE</load>
    <load>NTXDBE</load>
    <load>CDXDBE</load>
    <build name="DBFNTX">
      <data>DBFDBE</data>
      <order>NTXDBE</order>
    </build>
    <build name="DBFCDX">
      <data>DBFDBE</data>
      <order>CDXDBE</order>
    </build>
  </DatabaseEngines>
</Configuration>


//DbeLoad.Prg

FUNCTION ProcessConfig1(cFileName)  // Callback processing

LOCAL nXMLDoc,nActions

nXMLDoc := XMLDocOpenFile(cFileName)

nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/load",;
                            {|n,c,a,ch|handleLoad(n,c,a,ch)})

nActions := XMLDocSetAction(nXMLDoc, "//DatabaseEngines/build",;
                            {|n,c,a,ch|handleBuild(n,c,a,ch)})

IF nActions != 0
  XMLDocProcess(nXMLDoc)
  XMLDocResetAction(nXMLDoc)
ENDIF
XMLDocClose(nXMLDoc)

RETURN nil

/* Callback function to handle the DbeLoad action */
STATIC FUNCTION handleLoad(cTag,cContent,aMember,nH)

LOCAL aCH

XMLGetTag(nH,@aCH)
DbeLoad(aCH[XMLTAG_CONTENT],.F.)

RETURN (XML_PROCESS_CONTINUE)

/* Callback function to handle DbeBuild action */
STATIC FUNCTION handleBuild(cTag,cContent,aMember,nH)
```

```
LOCAL aCH :={}, nHChild,cName,cOrder,cData

cName := XMLGetAttribute(nH,"name")

nHChild := XMLGetChild(nH,"data")
XMLGetTag(nHChild,@aCH)
cData := aCH[XMLTAG_CONTENT]

nHChild := XMLGetChild(nH,"order")
XMLGetTag(nHChild,@aCH)
cOrder:= aCH[XMLTAG_CONTENT]

DbeBuild(cName,cData,cOrder)

RETURN (XML_PROCESS_CONTINUE)

* ==============

FUNCTION ProcessConfig2(cFileName) // Heirarchal processing

LOCAL oRootNode, aNodes, oNode, i, cName, cData, cOrder

oRootNode := DC_Xml2ObjectTree(cFileName)

aNodes := oRootNode:findNode({'DatabaseEngines','load'},.t.,.t.)

FOR i := 1 TO Len(aNodes)
  DbeLoad(aNodes[i]:content)
NEXT

aNodes := oRootNode:findNode({'DatabaseEngines','build'},.t.,.t.)

FOR i := 1 TO Len(aNodes)
  cName := aNodes[i]:getAttr("name")
  cData := aNodes[i]:findNode('data'):content
  cOrder := aNodes[i]:findNode('order'):content
  DbeBuild(cName,cData,cOrder)
NEXT

RETURN nil

* ==============

FUNCTION ProcessConfig3(cFileName) // XMLConfigLoad()

LOCAL oConfig, aDbes, abuild, i, cDbe, cName, cData, cOrder

oConfig := XmlConfigLoad('DbeLoad.Xml')

aDbes := oConfig:configuration:databaseengines:load
aBuild := oConfig:configuration:databaseengines:build

FOR i := 1 TO Len(aDbes)
  cDbe := aDbes[i]:getContent()
  DbeLoad(cDbe)
NEXT

FOR i := 1 TO Len(aBuild)
  cName := aBuild[i]:name
  cData := aBuild[i]:data
  cOrder := aBuild[i]:order
  DbeBuild(cName,cData,cOrder)
NEXT

RETURN nil
```

How to use the DC_XmlNode class

The DC_XmlNode class is used to build a heirarchal tree of node objects. There is 1 node for each XML tag. The tree is created by a function named DC_Xml2ObjectTree(). This function uses the Xbase++ Xml*() parser functions to parse the XML file or stream into the object tree.

```
Instance variables:

name        - The name of the XML tag.

content     - The content or data between the start tag and the end tag.

attrib      - A 2-dimensional array containing the name and value of all
the             attributes of the tag.

childList   - A 1-dimensional array containing the DC_XmlNode() objects
for all        child tags.

parent      - The DC_XmlNode() object that is the parent of this node.

cargo       - User defined cargo of any type.

caption     - Same as name.
```

Note: The childList, parent and caption iVars are compatible with Xbase parts naming convention so that the object tree can be viewed with the same utility function as Xbase Parts object trees.

```
Methods:

init( <cName>, [<xContent>], [<aAttrib>] ) -> self

This is the initialization routine.

  <cName> is the tag name.
  <xContent> is the tag contents.
  <aAttrib> is a 2-dimensional array of attributes.

---------

addChild( <oNode> ) -> self

This method adds a child node to this node.
<oNode> is an object of the DC_XmlNode() class.

---------

childList() -> aChildren

This method returns a 1-dimensional array containing the DC_XmlNode() objects for all
child tags.

---------

FindNode( <acTagName>, [<lMultiple>], [<lSearchSubs>] ) -> aoNode

This method is used to search child nodes in the tree for a specified node
or array of nodes based on its tag name.

<acTagName> is the the tag name of the child node to find. If <acTagName> is a
character string, then the search starts from the current node.  If
<acTagName> is an array of tag names, then the search starts at the first
child tag in the array.
```

```
<lMultiple> if .TRUE. will search for all nodes with <acTagName> and return
them in an array.

<lSearchSubs> if .TRUE. will search all sub nodes down the tree for
<acTagName>.

---------

GetAttr( <cAttrName>, [<lRemoveQuotes>] ) -> cValue

This method returns the value of an attribute as a character string.

<cAttrName> is the case-sensitive name of the attribute.

<lRemoveQuotes> will remove any quotation marks that are in the value.

--------

NodeToArray( <aArray> ) -> self

This method writes the entire object tree as a multi-dimensional array.

<aArray> is the array to write to.

--------

AsString() -> cString

This method writes the entire node and its child nodes to a properly formatted XML
string.

--------

WriteXML( <cFileName> ) -> lStatus

This method writes the entire node and its child nodes to a properly formatted XML
file.
```

Attaching Documents to XML

Any kind of document such as an image file, a PDF, a word document, or even a database can be embedded in an XML document. This is accomplished by encoding the document using Base64 encoding. Base64 is a binary to text encoding scheme that represents binary data in an ASCII string format.

Here is an example of embedding a database into an XML document and then extracting the database from the XML document.

These are the steps:

1. Load the database contents into a binary string.
2. Convert the string to a Base64 encoded string.
3. Build the XML with the encoded string.
4. Write the XML to a file.
5. Load the XML into an object tree using DC_Xml2ObjectTree()
6. Find the node containing the encoded data.
7. Convert the BASE64 encoded data back to a binary string.
8. Write the binary string to a new database.
9. Open the new database.

```
#INCLUDE "fileio.CH"

#Pragma Library("dcxml.lib")

#DEFINE CRLF Chr(13)+Chr(10)

FUNCTION Main()

LOCAL cString, nHandle, nSize, cEncodedString, cXml, oRoot, ;
      oDataNode, cDatabaseName, oNode

nHandle := FOpen('books.dbf')

nSize := FSeek( nHandle, 0, FS_END )

cString := Space(nSize)
FSeek( nHandle, 0, FS_SET )
FRead( nHandle, @cString, nSize )

cEncodedString := Bin2Base64(cString)

FClose( nHandle )

nHandle := FCreate('BOOKS.XML')

cXml := '<?xml version="1.0" encoding="UTF-8"?>' + CRLF
cXml += '<Database Name="Books2.dbf">' + CRLF
cXml += '  <MimeType>application/dbf</MimeType>'
cXml += '  <EncodedData>' + CRLF
cXml += cEncodedString + CRLF
cXml += '  </EncodedData>' + CRLF
cXml += '</Database>'

Fwrite( nHandle, cXml )
FClose( nHandle )

oRoot := DC_Xml2ObjectTree('BOOKS.XML')

oNode := oRoot:findNode('Database',,.t.)
cDatabaseName := oNode:getAttr('Name')

oDataNode := oRoot:findNode('EncodedData',,.t.)

cEncodedString := oDataNode:content
cString := Base642Bin(cEncodedString)

nHandle := FCreate(cDatabaseName)
FWrite( nHandle, cString )
FClose(nHandle)

USE (cDatabaseName)

RETURN nil
```

Attaching another XML or HTML document to an XML document

In an XML document, a **CDATA** section is a section of element content that is marked for the parser to interpret purely as textual data, not as markup. A CDATA section is merely an alternative syntax for expressing character data; there is no semantic difference between character data that manifests as a CDATA section and character data that manifests as in the usual syntax in which, for example, "<" and "&" would be represented by "<" and "&", respectively.

A CDATA section starts with the following sequence: <![CDATA[and ends with the first occurrence of the sequence:]]>

All characters enclosed between these two sequences are interpreted as characters, not markup.

In the below sample code, an HTML document is embedded within XML, the document is then loaded into a tree, the "document" node is found and the content is written out to a file.

```
#Pragma Library("dcxml.lib")

FUNCTION Main()

LOCAL cXml, oRoot, oDocument

TEXT INTO cXml WRAP
<?xml version="1.0" encoding="UTF-8"?>
<body>
  <filename>hello.htm</filename>
  <title>My HTML Document</title>
  <document>
  <![CDATA[
    <html>
      <head>
        <title>Hello World</title>
      </head>
      <body>
        <h2>My first web page</h2>
      </body>
    </html>
  ]]>
  </document>
</body>
ENDTEXT

oRoot := DC_Xml2ObjectTree(cXml)

oDocument := oRoot:findNode('document',,.t.)

MemoWrit('Hello.Htm',oDocument:content)
DC_SpawnUrl('hello.htm')

RETURN nil
```

Escaping XML Data

Adding control characters ('<','>', etc) into xml data could cause the parser to missunderstand the resulting data. The solution is to escape the control characters so that the parser can interpret them correctly as data, and not confuse them for markup.

Char	Escape String
<	<
>	>
"	"
'	'
&	&

These can be used within XML attributes, elements, and comments.
It is a good practice to always escape these characters when they appear in XML data, however this is not always required.

Examples:

Data	Xml Type	In XML
He said "OK"	Attribute	attributeName="He said "OK""
She said "You're right"	Attribute	attributeName="She said "You're right""
Smith&Sons	Attribute	attributeName="Smith&Sons"
if (age > 3 && age < 8)	Element	<MyElement>if (age > 3 && age < 8)</MyElement>
An XML Comment End with -->	Comment	<!-- An XML Comment End with --> -->
A & B	Comment	<!-- A & B -->

Var2Xml() and Xml2Var()

These two functions are orthogonal, meaning output of Var2XML() can be used as input for XML2Var() or vice versa. The following explaings the basic usage:

Code:

```
PROCEDURE main
LOCAL oData
oData := DataObject():new()
oData:array := {123, 'Text', Date()}
oData:codeblock := {|x| x + 1}
oData:character := "String with < & > ' " + Chr(34)
oData:date_var := CtoD('01/23/2016')
oData:logical := .t.
oData:number := 123.456
oData:undefined := NIL
cXML := Var2XML(oData)
oData1 := XML2Var( cXML )
oData1:number // 123.456
RETURN
```

Value of cXML:

```
<object classname="DataObject">
<array name="array">
<numeric>123</numeric>
<character>Text</character>
<date>20160830</date>
</array>
<codeblock name="codeblock">{|x| x + 1}</codeblock>
<character name="character">String with &lt; &amp; &gt; &#39; &quot;</character>
<date name="date_var">20160123</date>
<logical name="logical">true</logical>
<numeric name="number">123.456</numeric>
<undefined name="undefined"/>
</object>
```

Notes:
Only exported members of an object are added to the xml stream
Only members which exist are assigned to the object
var2xml/xml2var work with xml fragments not xml documents

Understanding Namespaces

This section about namespaces was borrowed from Tim Bray (http://www.xml.com/pub/a/1999/01/namespaces.html) because it was the simplest explanation of the reason for namespaces that I found.

by Tim Bray
January 19, 1999

Namespaces are a simple and straightforward way to distinguish names used in XML documents, no matter where they come from. However, the concepts are a bit abstract, and this specification has been causing some mental indigestion among those who read it. The best way to understand namespaces, as with many other things on the Web, is by example.

So let's set up a scenario: suppose XML.com wanted to start publishing reviews of XML books. We'd want to mark the info up with XML, of course, but we'd also like to use HTML to help beautify the display. Here's a tiny sample of what we might do:

```
<h:html xmlns:xdc="http://www.xml.com/books"
        xmlns:h="http://www.w3.org/HTML/1998/html4">
 <h:head><h:title>Book Review</h:title></h:head>
 <h:body>
  <xdc:bookreview>
   <xdc:title>XML: A Primer</xdc:title>
   <h:table>
    <h:tr align="center">
     <h:td>Author</h:td><h:td>Price</h:td>
     <h:td>Pages</h:td><h:td>Date</h:td></h:tr>
    <h:tr align="left">
     <h:td><xdc:author>Simon St. Laurent</xdc:author></h:td>
     <h:td><xdc:price>31.98</xdc:price></h:td>
     <h:td><xdc:pages>352</xdc:pages></h:td>
     <h:td><xdc:date>1998/01</xdc:date></h:td>
    </h:tr>
   </h:table>
  </xdc:bookreview>
 </h:body>
</h:html>
```

In this example, the elements prefixed with **xdc** are associated with a namespace whose name is**http://www.xml.com/books**, while those prefixed with **h** are associated with a namespace whose name is**http://www.w3.org/HTML/1998/html4**.

The prefixes are linked to the full names using the attributes on the top element whose names begin. **xmlns:**. The prefixes don't mean anything at all - they are just shorthand placeholders for the full names. Those full names, you will have noticed, are URLs, i.e. Web addresses. We'll get back to why that is and what those are the addresses of a bit further on.

Why Namespaces?

But first, an obvious question: why do we need these things? They are there to help computer software do its job. For example, suppose you're a programmer working for XML.com and you want to write a program to look up the books at Amazon.com and make sure the prices are correct. Such lookups are quite easy, once you know the author and the title. The problem, of course, is that this document has XML.com's book-review tags and HTML tags all mixed up together, and you need to be sure that you're finding the book titles, not the HTML page titles.

The way you do this is to write your software to process the contents of **<title>** tags, but only when they're in the**http://www.xml.com/books** namespace. This is safe, because programmers who are not working for XML.com are not likely to be using that namespace.

Attributes Too

Attributes, not just elements, can have namespaces. For example, let's use the HTML **STYLE** attribute to allow an HTML browser to display our book review:

```
<h:html xmlns:xdc="http://www.xml.com/books"
        xmlns:h="http://www.w3.org/HTML/1998/html4">
 <h:head><h:title>Book Review</h:title></h:head>
 <h:body>
  <xdc:bookreview>
   <xdc:title h:style="font-family: sans-serif;">
     XML: A Primer</xdc:title>
   <h:table>
    <h:tr align="center">
     <h:td>Author</h:td><h:td>Price</h:td>
     <h:td>Pages</h:td><h:td>Date</h:td></h:tr>
    <h:tr align="left">
     <h:td><xdc:author>Simon St. Laurent</xdc:author></h:td>
     <h:td><xdc:price>31.98</xdc:price></h:td>
     <h:td><xdc:pages>352</xdc:pages></h:td>
     <h:td><xdc:date>1998/01</xdc:date></h:td>
    </h:tr>
   </h:table>
  </xdc:bookreview>
 </h:body>
</h:html>
```

Beautification

That example above is, perhaps, kind of ugly, with all those prefixes and colons clutering up the tags. The Namespaces Recommendation allows you to declare a default namespace and leave out some prefixes, like this:

```
<html xmlns="http://www.w3.org/HTML/1998/html4"
      xmlns:xdc="http://www.xml.com/books">
 <head><title>Book Review</title></head>
 <:body>
  <xdc:bookreview>
   <xdc:title>XML: A Primer</xdc:title>
   <table>
    <tr align="center">
     <td>Author</td><td>Price</td>
     <td>Pages</td><td>Date</td></tr>
    <tr align="left">
```

```
      <td><xdc:author>Simon St. Laurent</xdc:author></td>
      <td><xdc:price>31.98</xdc:price></td>
      <td><xdc:pages>352</xdc:pages></td>
      <td><xdc:date>1998/01</xdc:date></td>
     </tr>
    </table>
   </xdc:bookreview>
  </body>
</html>
```

In this example, anything without a prefix is assumed to be in the **http://www.w3.org/HTML/1998/html4** namespace, which we're using as the namespace name for HTML (presumably, now that namespaces are official, the W3C will give HTML an official namespace name).

What Do Namespace Names Point At?

One of the confusing things about all this is that namespace names are URLs; it's easy to assume that since they're Web addresses, they must be the address of something. They're not; these are URLs, but the namespace draft doesn't care what (if anything) they point at. Think about the example of the XML.com programmer looking for book titles; that works fine without the namespace name pointing at anything. The reason that the W3C decided to use URLs as namespace names is that they contain domain names (e.g.**www.xml.com**), which work globally across the Internet.

Is That All There Is?

That's more or less all there is to it. The only purpose of namespaces is to give programmers a helping hand, enabling them to process the tags and attributes they care about and ignore those that don't matter to them.

Quite a few people, after reading earlier drafts of the Namespace Recommendation, decided that namespaces were actually a facility for modular DTDs, or were trying to duplicate the function of SGML's "Architectural Forms". None of these theories are true. The only reason namespaces exist, once again, is to give elements and attributes programmer-friendly names that will be unique across the whole Internet. Namespaces are a simple, straightforward, unglamorous piece of syntax.

Understanding SOAP

Although this session is supposed to be about XML, there is no discussion about XML that doesn't eventually turn into a discussion about SOAP because that is the most common method of transmitting XML across the internet. SOAP envelopes are created in XML.

SOAP (Simple Object Access Protocol) is a messaging protocol that allows programs that run on disparate operating systems (such as Windows and Linux) and also disparate languages (such as .NET and Xbase++) to communicate using Hypertext Transfer Protocol (HTTP) and its Extensible Markup Language (XML).

Since Web protocols are installed and available for use by all major operating system platforms, HTTP and XML provide a solution that allows programs running under different operating systems in a network to communicate with each other. SOAP specifies exactly how to encode an HTTP header and an XML file so

that a program in one computer can call a program in another computer and pass along information. SOAP also specifies how the called program can return a response. Despite its frequent pairing with HTTP, SOAP supports other transport protocols as well.

SOAP calls are much more likely to get through firewall servers, since HTTP is typically Port 80 compliant, where other calls may be blocked for security reasons. Since HTTP requests are usually allowed through firewalls, programs using SOAP to communicate can be sure that the program can communicate with programs anywhere.

Xbase++ 2.0 includes several classes that are ideal for creating a SOAP server. SOAPSERVER.PRG is the source for a SOAP server that uses the following Xbase++ classes:

HttpEndpoint
WebHandler
WebHandlerProcessor
FileHandlerProcessor

This server is basically a web server that handles all the file requests that are typical of any web server such as Apache and IIS but it also handles SOAP requests and creates appropriate responses. In the sample SoapServer.Prg there are only 2 SOAP methods:

getDateTime - Returns the date and time of the server
uploadXML – Uploads an XML document with authentication and saves the document to disk.

This is the XML Soap envelope sent by client for **getDateTime :**

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://donnay-software.com/soap">
   <soapenv:Header/>
   <soapenv:Body>
      <soap:GetDateTime/>
   </soapenv:Body>
</soapenv:Envelope>
```

This is the SOAP method code in Soap Server for **getDateTime :**

```
METHOD Soap:getDateTime()

RETURN ::response('GetDateTime', Dtos(Date()) + ' ' + Time())


METHOD Soap:response( cMethodName, cMessage )

LOCAL cResponse

DEFAULT cMessage := 'Message Received'

PRIVATE cMethodResponse := cMethodName + 'Response'
PRIVATE cMessageResponse := cMessage

TEXT INTO cResponse WRAP
<?xml version="1.0" ?>
<env:Envelope
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
     env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<env:Body>
<m:&cMethodResponse xmlns:m="http://donnay-software.com/soap">
  &cMessageResponse
</m:&cMethodResponse>
</env:Body>
</env:Envelope>
ENDTEXT

RETURN cResponse
```

This is the **getDateTime** XML response that is sent back to the client program:

```
<env:Envelope env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <env:Body>
      <m:GetDateTimeResponse xmlns:m="http://donnay-software.com/soap">20160803
17:32:01</m:GetDateTimeResponse>
   </env:Body>
</env:Envelope>
```

This is the XML Soap envelope sent by client for **uploadXML :**

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://xb2.net/soap">
   <soapenv:Header/>
   <soapenv:Body>
      <soap:UploadXML>
         <soap:UserName>Alaska</soap:UserName>
         <soap:Password>PolarBear</soap:Password>
         <soap:XmlDocument><Message>Hello World</Message></soap:XmlDocument>
      </soap:UploadXML>
   </soapenv:Body>
</soapenv:Envelope>
```

This is the SOAP method code in Soap Server for **uploadXML :**

```
METHOD Soap:UploadXML( oMainNode )

LOCAL oNode, cResponse, cString, cMethod := 'UploadXML'

oNode := oMainNode:findNode('UserName',,.t.)
IF oNode == NIL
  RETURN ::errorResponse( cMethod,'No "UserName" Tag in Soap Request')
ENDIF
IF oNode:content # 'Alaska'
  RETURN ::errorResponse( cMethod,'Invalid UserName')
ENDIF

oNode := oMainNode:findNode('Password',,.t.)
IF oNode == NIL
  RETURN ::errorResponse( cMethod,'No "Password" Tag in Soap Request')
ENDIF
IF oNode:content # 'PolarBear'
  RETURN ::errorResponse( cMethod,'Invalid Password')
ENDIF

oNode := oMainNode:findNode('XmlDocument',,.t.)

IF oNode == NIL
  RETURN ::errorResponse( cMethod,'No "XMLDocument" Tag in Soap Request')
ENDIF

cString := oNode:asString()

MemoWrit( Dtos(Date())+Strtran(Time(),':','')+'.Xml',cString )
```

```
RETURN ::response( cMethod, 'XML Successfully received' )
```

This is the **uploadXML** XML response that is sent back to the client program:

```
<env:Envelope env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <env:Body>
      <m:UploadXMLResponse xmlns:m="http://donnay-software.com/soap">XML Successfully
received</m:UploadXMLResponse>
   </env:Body>
</env:Envelope>
```

Testing of your SOAP methods can be handled with a free utility called SoapUI. It is available for download at https://www.soapui.org/downloads/soapui.html.

Understanding WSDL

The **Web Services Description Language** (WSDL pronounced wizdal) is an XML based definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameter it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a method definition in a programming language.

Many web servers that support web services such as SOAP also publish a WSDL file which is used to format client requests. Here is an example of a WSDL file that would be used to define the SOAP methods (getDateTime and uploadXML) supported by SoapServer.Prg:

```
<?xml version="1.0" encoding="UTF-8"?>
<w:definitions
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:m="http://xb2.net/soap"
   targetNamespace="http://xb2.net/soap"
   xmlns:w="http://schemas.xmlsoap.org/wsdl/">

   <w:documentation xmlns:w="http://schemas.xmlsoap.org/wsdl/">Xbase++ Web Service
Interface</w:documentation>

   <w:types>
     <s:schema elementFormDefault="qualified" targetNamespace="http://xb2.net/soap">

       <s:element name="GetDateTime">
         <s:complexType />
       </s:element>
       <s:element name="GetDateTimeResponse">
         <s:complexType>
           <s:sequence>
             <s:element minOccurs="0" maxOccurs="1" name="DateTime" type="s:string"/>
```

```
            </s:sequence>
          </s:complexType>
        </s:element>

        <s:element name="UploadXML">
          <s:complexType>
            <s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="UserName" type="s:string"/>
              <s:element minOccurs="0" maxOccurs="1" name="Password" type="s:string"/>
              <s:element minOccurs="0" maxOccurs="1" name="XmlDocument"
type="s:string"/>
            </s:sequence>
          </s:complexType>
        </s:element>
        <s:element name="UploadXMLResponse">
          <s:complexType>
            <s:sequence>
              <s:element minOccurs="0" maxOccurs="1" name="Result" type="s:string"/>
            </s:sequence>
          </s:complexType>
        </s:element>

      </s:schema>
    </w:types>

    <w:message name="GetDateTimeSoapIn">
       <w:part name="parameters" element="m:GetDateTime"/>
       </w:message>
    <w:message name="GetDateTimeSoapOut">
       <w:part name="parameters" element="m:GetDateTimeResponse"/>
       </w:message>

    <w:message name="UploadXMLSoapIn">
       <w:part name="parameters" element="m:UploadXML"/>
       </w:message>
    <w:message name="UploadXMLSoapOut">
       <w:part name="parameters" element="m:UploadXMLResponse"/>
       </w:message>

    <w:portType name="SoapServiceSoap">
       <w:operation name="GetDateTime">
          <w:input message="m:GetDateTimeSoapIn"/>
          <w:output message="m:GetDateTimeSoapOut"/>
       </w:operation>
       <w:operation name="UploadXML">
          <w:input message="m:UploadXMLSoapIn"/>
          <w:output message="m:UploadXMLSoapOut"/>
       </w:operation>
    </w:portType>

    <w:binding name="SoapServiceSoap" type="m:SoapServiceSoap">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>

       <w:operation name="GetDateTime">
          <soap:operation soapAction="GetDateTime" style="document"/>
          <w:input><soap:body use="literal"/></w:input>
          <w:output><soap:body use="literal"/></w:output>
       </w:operation>

       <w:operation name="UploadXML">
          <soap:operation soapAction="UploadXML" style="document"/>
          <w:input><soap:body use="literal"/></w:input>
          <w:output><soap:body use="literal"/></w:output>
       </w:operation>

    </w:binding>

    <w:service name="SoapService">
       <w:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
Xbase++ Test Web Service Interface</w:documentation>
```

```
      <w:port name="SoapServiceSoap" binding="m:SoapServiceSoap">
         <soap:address location="http://localhost:81/soap/services"/>
      </w:port>

   </w:service>

</w:definitions>
```

Conclusion

Xbase++ Is a very good language for working with XML. It's class architecture is ideal for writing XML classes. It's internet classes such as HttpEndPoint, WebHandler, LoadFromUrl are ideal for writing SOAP servers and SOAP clients. Learning to work with and understand XML will benefit the Xbase++ developer in many ways.