

oXML

The XML component for Omnis Studio

TigerLogic Corporation

October 2009

10-102009-01

The software this document describes is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. Names of persons, corporations, or products used in the tutorials and examples of this manual are fictitious. No part of this publication may be reproduced, transmitted, stored in a retrieval system or translated into any language in any form by any means without the written permission of TigerLogic.

© TigerLogic Corporation, and its licensors 2009. All rights reserved.
Portions © Copyright Microsoft Corporation.

© 1999-2009 The Apache Software Foundation. All rights reserved.
This product includes software developed by the Apache Software Foundation
(<http://www.apache.org/>).

OMNIS® and Omnis Studio® are registered trademarks, and Omnis 7™ is a trademark of TigerLogic Corporation.

Microsoft, MS, MS-DOS, Visual Basic, Windows, Windows 95, Win32, Win32s are registered trademarks, and Windows NT, Visual C++ are trademarks of Microsoft Corporation in the US and other countries.

Apple, the Apple logo, AppleTalk, and Macintosh are registered trademarks and MacOS, Power Macintosh and PowerPC are trademarks of Apple Computer, Inc.

IBM and AIX are registered trademarks of International Business Machines Corporation.

UNIX is a registered trademark in the US and other countries exclusively licensed by X/Open Company Ltd.

Sun, Sun Microsystems, the Sun Logo, Solaris, Java, and Catalyst are trademarks or registered trademarks of Sun Microsystems Inc.

HP-UX is a trademark of Hewlett Packard.

OSF/Motif is a trademark of the Open Software Foundation.

Acrobat is a trademark of Adobe Systems, Inc.

ORACLE is a registered trademark and SQL*NET is a trademark of Oracle Corporation.

SYBASE, Net-Library, Open Client, DB-Library and CT-Library are registered trademarks of Sybase Inc.

INFORMIX is a registered trademark of Informix Software, Inc.

EDA/SQL is a registered trademark of Information Builders, Inc.

CodeWarrior is a trademark of Metrowerks, Inc.

Other products mentioned are trademarks or registered trademarks of their corporations.

Table of Contents

ABOUT THIS MANUAL	5
FURTHER INFORMATION	5
INTRODUCTION	6
WHAT IS XML?	6
<i>Elements</i>	6
<i>Attributes</i>	8
<i>Entities</i>	8
<i>DTDs</i>	8
<i>Schemas</i>	9
<i>XML Parser</i>	10
<i>Displaying XML documents</i>	10
<i>What are the benefits of XML?</i>	11
ABOUT THE DOM	12
<i>What is the DOM?</i>	12
<i>Documents and Nodes</i>	13
USING OXML	14
INSTALLATION	14
SERIALIZATION	15
<i>Deployment Licensing</i>	15
CREATING A DOCUMENT OBJECT	16
<i>Creating an object variable</i>	16
<i>Inspecting an object variable</i>	17
DOCUMENT OBJECTS IN OXML.....	18
MANIPULATING XML DOCUMENTS.....	19
<i>Loading an XML document</i>	19
<i>Getting the document root element</i>	21
<i>Getting the attributes of an element</i>	21
<i>Adding children to a node</i>	22
<i>Saving a document</i>	24
<i>Using Lists with XML</i>	25
<i>Using Tree lists with XML</i>	26
<i>Document Templates</i>	29
<i>Character Sets and Unicode</i>	29
CREATING XML DOCUMENTS	29
<i>Creating an XML Document</i>	29
<i>Saving the XML File</i>	31

REFERENCE.....	33
COMMON.....	33
<i>Properties</i>	33
ATTRIBUTE.....	34
CDATASECTION.....	34
COMMENT	35
DOCUMENT	35
DOCUMENT FRAGMENT.....	40
DOCUMENTTYPE	41
ELEMENT.....	42
ENTITY	44
NAMEDNODEMAP.....	45
NODE.....	46
NODELIST.....	48
NOTATION	48
PROCESSINGINSTRUCTION.....	49
TEXT	49
CONSTANTS	50
FUNCTIONS.....	51
<i>OXML.\$base64decode()</i>	51
<i>OXML.\$base64encode()</i>	51
<i>OXML.\$formatbinaschar()</i>	51
<i>OXML.\$maybexml()</i>	52
<i>OXML.\$striphttpheader()</i>	52
INDEX	53

About This Manual

This manual describes the Omnis Studio XML object (oXML), an external component which allows you to parse and manipulate XML documents in Omnis using the standard Document Object Model (DOM) API. This manual contains a basic introduction to XML and the DOM. This manual does not provide an exhaustive description of XML or the DOM, since this can be gained from many other sources, such as those listed below.

Further Information

For further information about XML, you should surf the internet for XML-related web sites, and read one or two of the many books available on the subject. Here are one or two resources we found very useful:

- ❑ **XML and DOM standards**
www.w3.org has the official XML standards and a lot of good general information; also includes a full definition of the DOM API. A look at the DOM Level 2 definition is very useful, whereas spending a lot of time on the XML specification is unnecessary.
- ❑ **Information and Tutorials**
www.xml.com and www.xml101.com contain many XML-related articles, tutorials and news.
www.w3schools.com & www.webmonkey.com both have background information and some useful online tutorials.
- ❑ **Essential XML for Web Professionals**, by Dan Livingstone, Prentice Hall PTR; ISBN: 0130662542
Provides an excellent introduction to XML and includes sample files and tutorials on an accompanying web site: www.phptr.com/essential/xml
Contains the XML 1.0 specification, and a very understandable chapter about the DOM (Document Object Model) which oXML uses to access XML documents.
- ❑ **Professional XML (Programmer to Programmer)** 2nd Edition, by Mark Birbeck et al, Wrox Press Inc; ISBN: 1861005059
This is an extensive guide for application developers and programmers going well beyond the basics. It's good on XML syntax and is a good reference guide for programmers, and assumes you are already familiar with XML.

Introduction

The interface to XML documents is implemented in Omnis Studio using the standard Document Object Model (DOM) API as an external component which must be instantiated via an Omnis object variable. The oXML component addresses the most basic XML requirement, namely the ability to parse and extract information from an XML document, and to generate new XML documents. The oXML component allows you to parse and manipulate XML documents using a standard set of methods provided by the DOM level 2 API, plus some additional methods that speed up the process of building a document. The oXML component also allows you to display an XML document in the tree list component, which is well suited to displaying the hierarchical structure contained in XML documents. The oXML is supported in Omnis Studio 3.1 or later under Windows, Unix and Mac OS.

To use oXML to access your XML documents, you need a working knowledge of XML and the DOM. This section provides a short introduction to XML and the DOM. If you are already familiar with these technologies, and/or you have read one of the many sources of information about XML and DOM, then please skip this section.

What is XML?

XML (eXtensible Markup Language) allows you to store, exchange and display data or information in a structured and efficient way. In this respect it is no different from most existing data formats, except that XML provides a higher degree of standardization and flexibility than many other proprietary technologies, opening up many new and exciting opportunities in business computing and information technology. XML has already revolutionized content management, information publishing, and news syndication, as well as other B2B markets, while the adoption of XML across many other industry sectors seems to be gathering pace.

XML allows you to store structured documents or data as text and provides you with a way of manipulating, transforming, and presenting your data in many different formats. For example, information or data stored in an XML document can be displayed in a web browser using a Cascading Style Sheet (CSS). In addition, when XML documents are stored in a database they can be queried and retrieved much like any other data source.

Elements

XML is very much like HTML, but it differs in one or two important ways. Like HTML, XML uses tags to define the "elements" (content or data holders) within a document, but unlike HTML, XML tags only describe the data or content, they do not contain any information about the display or formatting of the content or document as a whole. Each element must have a start and an end tag, and tag names are case-sensitive.

HTML conforms to a standard set of tags, whereas XML element names can be anything you like providing a better description of each piece of data or the content in your document. For example, to create a file to store the contents of a bookstore you can create an element called `<bookstore>` to contain the information about all the books. XML documents are often described as having “meta-data” since the information in the tags describes the data within the tag itself. In this case, someone looking at the file containing the tag `<bookstore>` can see immediately that the information relates to a bookstore.

Elements within a document are often nested in a hierarchical structure, building a more detailed or structured picture of the thing or things being described in the document. Therefore, individual elements are referred to as “nodes”. The top level element in a document is called the “root node”, which has an ID of 0, and all elements inside it are called “child nodes” which have unique IDs identifying them. Carrying on the book example, the `<bookstore>` element or root node could contain elements for `<book>`, `<title>`, `<author>`, `<publisher>`, and `<isbn>`, plus you can further describe the `<author>` element using `<firstname>` and `<lastname>` child nodes. An XML document with these elements or nodes would have the following structure:

```
<?xml version="1.0"?>
<bookstore>
  <book>
    <title>Essential XML for Web Professionals</title>
    <author>
      <firstname>Dan</firstname>
      <lastname>Livingstone</lastname>
    </author>
    <publisher>Prentice Hall PTR</publisher>
    <isbn>0130662542</isbn>
    <price>34.99</price>
  </book>
  <book>
    <title>Professional XML (Programmer to Programmer)</title>
    <author>
      <firstname>Mark</firstname>
      <lastname>Birbeck</lastname>
    </author>
    <publisher>Wrox Press, Inc</publisher>
    <isbn>1861005059</isbn>
    <price>59.99</price>
  </book>
</bookstore>
```

Note the `<firstname>` and `<lastname>` elements are nested inside the `<author>` element, while all sub-nodes are contained in the `<bookstore>` root node. Also note the obligatory

XML declaration at the beginning of the document which defines the XML version of the document; this is a processing instruction that gets sent to the XML parser.

Attributes

Like HTML, elements can have "attributes" (properties) that further describe the element, but again they do not provide any information about the display of the data. For example, the `<book>` element in our sample xml above could have the attribute "genre" which is written like this:

```
<book genre="Computing">
```

Note that genre is a general characteristic of a book and is therefore considered an attribute of a book (i.e. many books may be in the same genre), whereas the title of a book is unique to each book and is therefore described in an element as part of an individual book.

Entities

Entities let you represent a single character, a number of characters, or a string of words using a short alias name. There is a range of ISO approved entities that have reserved name and number codes to represent specific characters, such as `&` for ampersand, `<` for lesser than, `>` for greater than, `"` for double quotation mark, `'` for apostrophe, and `€` for the Euro symbol, and so on. You can also define your own custom entities in your XML documents in the document template, either internally quoted in the DTD (Document Type Definition: see below) or they can be listed in an external file. For example, you could represent a publisher name by declaring `<!ENTITY ph "Prentice Hall PTR">`, therefore writing the publisher name as `&ph;` in the body of your document.

Entities that hold text, like those described above, are called *parsed entities*. You can also create entities for non-text data or files, such as image files, video, binary files, or even other applications, and these are called *unparsed entities*, but they are also referred to as *notations*.

DTDs

When a document has all the correct start and end tags and is properly nested, it is described as "well-formed". XML documents are processed through an XML parser which checks for correct syntax or "well-formedness". Documents can be further *validated* against a template or *Document Type Definition* (DTD), or a schema. A DTD is itself a text document which contains a description of the elements and entities for a particular type of XML document, in other words, it specifies what type of data or content the document can contain. The DTD would contain a list of elements allowed in the XML document, defining the name and data type for each element. The DTD for an XML document can be included inline, as part of the XML document itself, or it can be a separate file referenced in the XML document.

```
<!DOCTYPE BOOKS [  
<!ENTITY PH "Prentice Hall PTR">  
<!ELEMENT book (TITLE,AUTHOR,PUBLISHER,ISBN)>  
<!ELEMENT title (#PCDATA)>  
<!ELEMENT author (FIRSTNAME,LASTNAME)>  
<!ELEMENT firstname (#PCDATA)>  
<!ELEMENT lastname (#PCDATA)>  
<!ELEMENT publisher (#PCDATA)>  
<!ELEMENT isbn (#PCDATA)>  
>
```

Most of the time you will need to use an industry standard DTD, rather than creating your own. Using standard DTDs ensures the portability of your data or documents across many different applications and between organizations.

Schemas

Increasingly, DTDs are being superseded by *schemas*, or *w3c schemas* as they are sometimes called. oXML supports the use of schemas to validate XML documents. Like DTDs, schemas define the structure and types of data allowed in documents but they provide greater control over the structure and types of data in your XML documents. Schemas use XML *namespaces* to define the elements and attributes in your XML documents. Namespaces are unique names that identify element types and attribute names.

Schemas are themselves written in XML so you can create and manipulate them using oXML. Schemas are more powerful than DTDs since you can define the type and constraints on the data in your documents. Schemas can contain a number of built-in data types, such as, xs:string, xs:decimal, xs:date, xs:anyURI, and you can create your own custom types. Like DTDs, schemas can be referenced externally in your XML documents.

The differences between DTDs and Schemas become apparent when you compare one with the other, for example, the following DTD called note.dtd describes the structure of a very simple XML document.

```
<!ELEMENT note (to, from, heading, body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>
```

The DTD defines the ‘note’ root node as having four child nodes (to, from, heading, body). The DTD is placed in the XML document itself or referenced externally.

A simple schema called `note.xsd` can be used to define the same structure:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The schema defines the ‘note’ *complex element* that contains four *simple elements* (to, from, heading, body).

XML Parser

An XML parser or processor is a software module that checks your documents for well-formedness and performs validation against a DTD. The XML parser provided with oXML, called Xerces (called ‘xerces-c_1_6_0.dll’ under Windows), is a validating parser that allows you to read and write documents as well as perform validation against a DTD or schema. The Xerces parser should be placed in the root of your Omnis Studio folder, i.e. in the same location as the Omnis.exe program. Under Mac OS X the parser library is called Xerces.Classic.Lib and it is bound into the oXML component.

Displaying XML documents

Most recent browsers, including Internet Explorer 5, will display XML documents in a collapsible/expandable format. For most purposes though, you need to extract data from your XML documents for subsequent data processing, or enumerate an entire document in order to build a list for display in an Omnis tree list.

The display or transformation of the XML data is handled at the time of delivery when the document is retrieved from a document store and displayed on a client machine. The idea of XML is to store your data in a raw but structured state, allowing you to query and present it in many different ways as and when required.

What are the benefits of XML?

The business benefits of using XML and XML-based systems are well documented in the IT industry and media. XML, or rather technologies that use XML as their basis, promise to provide the IT industry with greater standardization, interoperability, efficiency, and present the potential for many new technologies. If you are an application developer, you will no doubt be asked some time in the future to create applications that will “handle XML”.

❑ **Platform Independent and Reusable**

XML is machine and platform independent so it can be exchanged between one system or network and another. Plus, once information is in XML format it can be reused for many different purposes for digital and printed publication.

❑ **Worldwide Standard**

XML is a standard language defined and ratified by the W3C consortium so it is not controlled or owned by any one company. This ensures the future of XML as an open standard employed by the whole IT industry.

❑ **Information exchange**

Since XML is an agreed standard it affords a high degree of information exchange, in particular between networks, businesses and other interdependent organizations.

❑ **New Business opportunities**

The standardization and flexibility of XML mean that many existing business problems can be solved more efficiently, while many new business opportunities will arise that take advantage of XML. For example, XML has already revolutionized Content management, publishing & news syndication, and will transform many other areas of business, particularly those suited to automation.

About the DOM

To use oXML to access your XML documents, you need some knowledge of the DOM. Like XML, the DOM API is well documented in print and web form so consult these external sources for further information.

What is the DOM?

The Document Object Model (DOM) is an API that allows you to build documents, navigate their structure, and add, modify, or delete elements and content. To quote from www.w3c.org, the “Document Object Model (DOM) is an application programming interface (API) for XML [and HTML] documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term “document” is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.”

The oXML component uses the DOM Level 2 API to access XML documents, as defined on the W3C web site. It is built upon source code provided by the Xerces project, available as part of the Apache XML project: please see their web site for background information (<http://xml.apache.org>). For our purposes, the DOM provides a platform-independent interface to XML documents so that Omnis developers can use Omnis code and the notation to navigate and manipulate XML documents. DOM treats an XML document as a hierarchy of nodes, arranged in a tree structure, and accessible via its API and its methods.

The methods available in oXML closely match those defined by the DOM, so for a fuller explanation of the DOM API and its interfaces and methods you should consult the www.w3.org web site, or a good XML book or reference guide. The following URL has a definition of the DOM Level 2 API:

<http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html>

Documents and Nodes

The DOM represents a document as a tree of nodes or objects. Each node represents a different part of the document, hence a node can be one of a number of different types of node. In addition, each node or object type can have children, but only certain types of children are allowed for each type of node. The following table shows you what objects are returned (if any) when you query the children of a node in the document tree.

Node type	Possible Children
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	no children

See the Reference section later in this manual for a complete list of properties and methods for each type of node object.

Using oXML

This section shows you how to manipulate and create XML documents using the oXML component.

Installation

The oXML package is available for all the supported platforms including all Windows, Unix, and Mac OS platforms. The component and associated files are compressed together in a single file, either a .zip, .taz, or .sit depending on the platform. The oXML package contains a number of different components which must be placed in the appropriate folders inside your main Omnis Studio folder.

The file names or extension names of the files vary for different platforms, but the files are essentially the same and should be placed in the following folders:

- ❑ **Xcomp/oxml.dll**
place in the XCOMP folder under the main Omnis folder
- ❑ **xerces-c_1_6_0.dll** (Win)
the XML parser is placed in the root of the main Omnis folder (i.e. the same location as Omnis.exe); under Mac OS X the file Xerces.Classic.Lib is bound into the oXML component so is not required in the Omnis folder

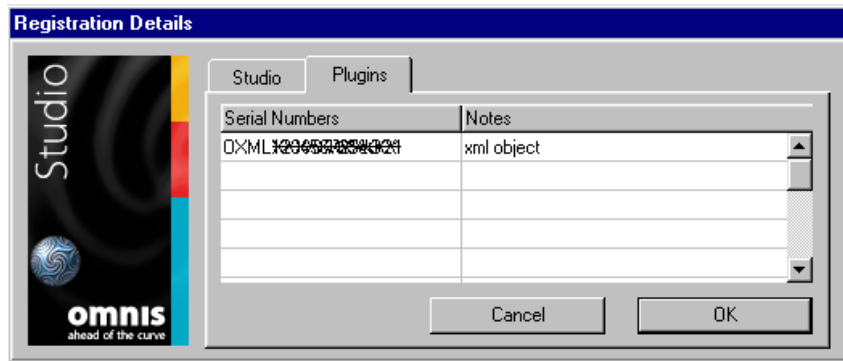
Note oXML is supported in Omnis Studio 3.1 or later, but developers using Omnis Studio 3.1 only should note that the tree component was modified for use with oXML, therefore you should contact Omnis Support for a copy of the new tree component to run with Studio 3.1. Developers using Omnis Studio 3.2 onwards, or Studio 4.x, will have the new tree component and do not need to do anything.

Serialization

Having copied the oXML files to the appropriate locations, you need to restart Omnis and enter your oXML serial number. Each development copy of Omnis Studio must be serialized with a different plugin serial number to enable the oXML component.

To serialize oXML

- Start Omnis Studio and select the **Change Serial Number** option in the **Tools** menu



- In the Change Serial Number dialog box, click on the **Plugins** tab and enter your oXML serial number in the 'Serial Numbers' column
- Click on OK

Note you have to restart Omnis Studio for the oXML serial number to take effect.

Deployment Licensing

You can deploy the oXML component free of charge, but each Runtime or Server copy of Omnis Studio that you deploy must be serialized to enable the component. When end-users install the Omnis Studio Runtime they must enter their runtime number and the oXML plug-in serial number via the Plugins tab.

Alternatively, you can add your oXML plug-in serial number to the Serial.txt file which is found in the root of the Runtime or Server product tree. Add the oXML serial number to the [plug-ins] section of this file. You can copy the entry from your development Serial.txt file to get the exact syntax. The format is as follows:

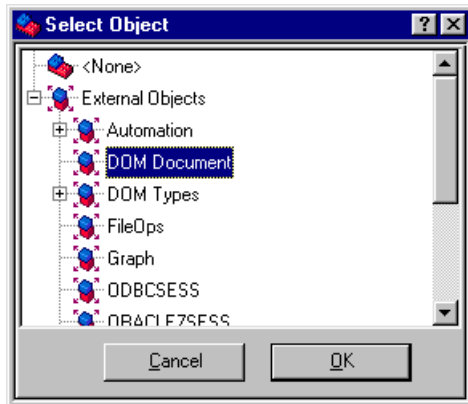
```
[Plugins]
PLUGIN=OXML12121212112
```

Creating a Document Object


The oXML component is an *external object* which is a type of external component that contain *methods* that you can use by instantiating an object variable based on the external object. The oXML component is stored in a component library, called Oxml.dll under Windows, and should be placed in the XCOMP folder. The oXML component is always loaded by default so there's no need to load it via the External Components option in the Component Store.

Creating an object variable

You can add a new XML object in the method editor by inserting a variable of type Object and using the subtype column to select the XML Document object. You can click on the subtype droplist and select the XML object from the Select Object dialog. You can ignore the group of DOM Types: you must create a document object to access the whole of an XML document and use the object's methods to return the different parts or elements in the document.



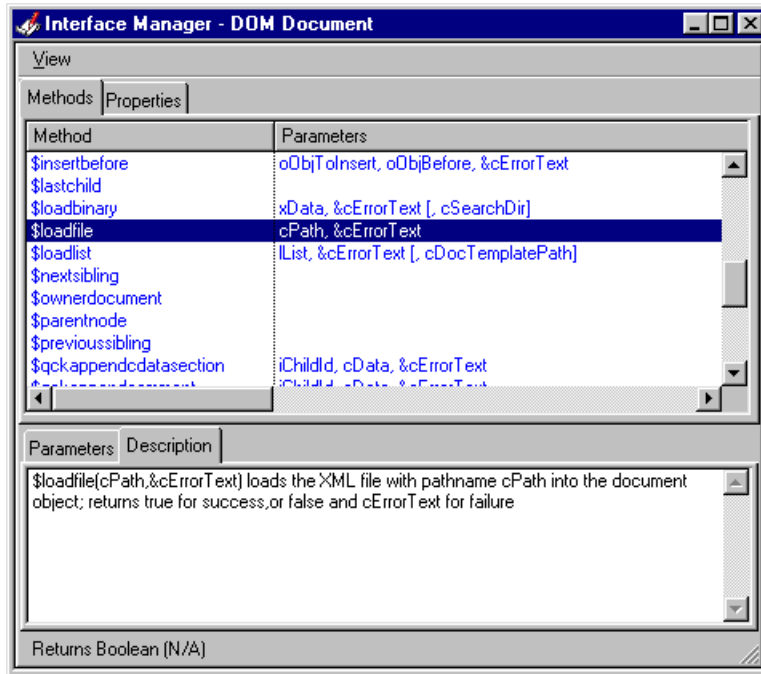
An object icon plus the type “DOM Document” will appear in the variable subtype cell showing the type of object.

	Variable	Type	Subtype	Init.Val/Calc
3	iEnt	Boolean	N/A	
4	iValidate	Boolean	N/A	kTrue
5	path	Character	10000000	
6	xml	Object	 DOM Document	

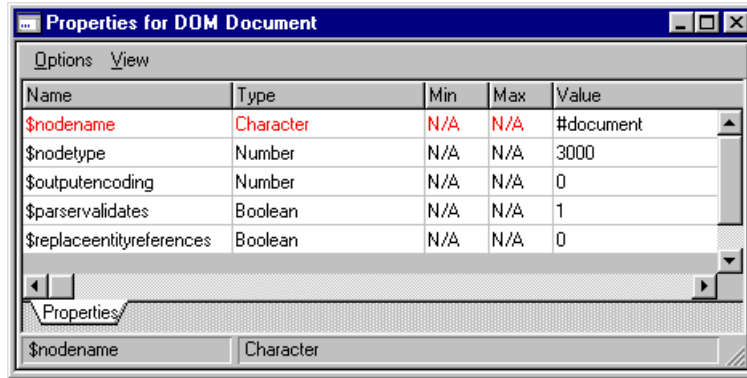
Task \ Class \ Instance \ Local \ Parameter

Inspecting an object variable

When an instance of the external object has been constructed (in an open window or form), you can inspect its properties and methods during development using the Interface Manager.



You can drag a method of the XML object from the Interface Manager into the command parameters box of the Do command. For example, when you inspect a DOM Document object (as shown), the Interface Manager will list the methods of a document object as well as the general properties of a node, since the root node is at the head of the document. To load a specified XML document you can use the \$loadfile() or \$loadbinary method (see below in the section Loading an XML document for the code). While debugging your code, you can inspect an object variable by right-clicking on the variable and selecting the Variable [varname] option. The following shows an object variable containing a document.



Document Objects in oXML

In the DOM, documents have a logical structure which is like a tree, that is, all the elements and objects in the document are arranged in a hierarchical structure. Each object in a document is treated as a “node”. The objects or nodes in a document can be one of a number of different types: an element, a comment, an attribute, text, or an entity. The oXML component contains a number of different objects, to support the different objects defined by the DOM.

The key object in oXML is the “DOM Document” object, which represents an XML document. You can use DOM Document as the subtype of an object variable, in order to use oXML and access its methods to manipulate your documents.

Having returned an XML document into your object variable, you can use various methods to return the objects within the document. For example, the \$documentelement() method returns a DOM Element object that, in this particular case, represents the root of the XML document (see below the section Getting the document root element for the code). You can return and manipulate the following objects:

- DOM Attribute**
an attribute or property of an element
- DOM CDATASection**
the CDATA section of an attribute definition
- DOM Comment**
provides access to the content of a comment
- DOM DocumentFragment**
allows you to load part of a document into memory rather than the whole document
- DOM DocumentType**
a list of document types within a document

- ❑ **DOM Element**
an element within a document; provides access to an element's attributes
- ❑ **DOM Entity**
an entity within a DTD
- ❑ **DOM EntityReference**
the representation of an entity
- ❑ **DOM NamedNodeMap**
a collection of unordered nodes within a document
- ❑ **DOM NodeList**
an ordered list of nodes within a document
- ❑ **DOM Notation**
an unparsed or non-textual entity within a DTD, or the formal declaration of Processing Instruction target
- ❑ **DOM ProcessingInstruction**
a special element or tag that provides instructions parsed to an external application, e.g. the XML version declaration at the beginning of a document
- ❑ **DOM Text**
the text or content of an element or attribute

These objects are never used directly as the subtype of an object variable. Instead, they are returned by methods of the oXML object.

Manipulating XML Documents

Having created and instantiated an object variable, based on oXML, you can use its methods to load an XML document, enumerate the different parts or elements of the document, and display it in an Omnis tree list component. The following sections show how you can do this using Omnis methods.

Loading an XML document

You can use the \$loadfile() or \$loadbinary() method to load an XML document into the document object (variable).

- ❑ **\$loadfile(cPath,&cErrorText)**
loads the XML file with pathname cPath into the document object, and returns true for success, or false and cErrorText for failure.
- ❑ **\$loadbinary(xData,&cErrorText)**
loads a document stored in the binary variable xData into the document object; returns true for success, or false and cErrorText for failure.

When the `$loadfile()` method is executed the specified document is loaded into the object object/variable. In effect, this method loads the whole document, that is, a DOM Document representing the document. The document object has the general properties of a node (`$nodename=#Document` and `$nodetype=kXMLNodeDocument`) together with the properties of a document object `$parservalidates`, `$replaceentityreferences`, and `$outputencoding`, as well as many other methods for manipulating or traversing the document tree.

The following method can be placed behind a button and be used to prompt the user to select an XML document. The method then calls a class method to build the tree corresponding to the structure of the document selected by the user.

```
; Method '$event' for Load XML button
; create instance vars: path (Char), xml (Object, DOM Document),
  iValidate (Bool), iEnt (Bool), error (Num, Long Int), errorText
  (Char)
On evClick
  Calculate path as
  Do FileOps.$getfilename(path,"Select XML file to parse")
  ; prompts the user for an XML file name and location
  If path<>' '
    Calculate $cinst.$title as path
    Calculate xml.$parservalidates as iValidate ;; optional
    Calculate xml.$replaceentityreferences as iEnt ;; optional
    Do xml.$loadfile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    Else
      Do method $buildtree ;; see below
    End If
  End If
```

Note that the variables `iValidate` and `iEnt` can be added to your window or application (these preferences can be assigned to check boxes on a window) and used to force the XML parser to validate your document and resolve all entities.

Getting the document root element

Having loaded the XML document into the document object, you can get the root element using the `$documentelement()` method; the method has no parameters.

- ❑ **`$documentelement()`**
returns the DOM Element object representing the root of this XML document.

The following method prepares the window tree list, gets the root element from the document object and calls another method to build up a complete tree containing all the sub-nodes in the document.

```
; Method '$buildtree'
; create local variables nodetext (Char), obj (Object), tree (Item
  ref), treenode (Item ref)
Set reference tree to $cinst.$objs.tree
Do tree.$clearallnodes()
Calculate obj as xml.$documentelement()
; the root element is returned and placed in 'obj'
Do method $getelementtext (obj) Returns nodetext ;; see below
Set reference treenode to tree.$add(nodetext)
Do method $addchildren (obj,treenode) ;; see below
```

Getting the attributes of an element

When you have placed an element, such as the root element, into an object variable you can get its text value held in the `$tagname` property and access its attributes, if the element has any, using the `$attributemap()` method.

- ❑ **`$attributemap(&cErrorText)`**
returns a named node map object listing the attributes of the element, or NULL and `cErrorText` if an error occurs; the named node map contains an unordered list of attributes belonging the element

Having created the list of attributes (a named node map) for an element you can step through the list using the `$item()` method in a For loop to extract each attribute.

- ❑ **`$item(iIndex)`**
returns an attribute object referenced by `iIndex` from the name node map; indexing starts at zero; a bad index results in a NULL return value.

The following method gets the text value of the element passed to it and, assuming the element has attributes, adds the attributes in 'name=value' pairs. The properties \$attname and \$attvalue give you the name and value of an attribute.

```
; Method '$getelementtext'  
; create parameter var element (Object)  
; create local vars att (Object), attlist (Object), k (Long int),  
  nodetext (Char)  
Calculate nodetext as con('<',element.$tagname,'>')  
; returns the element or tag name in nodetext  
Calculate attlist as element.$attributemap  
; builds a 'namednodemap' or list of object attributes; this is  
  empty if there are no attributes and code skips the For loop,  
  otherwise loop adds all attributes of tag  
For k from 0 to attlist.$length-1 step 1  
  ; $length is the number of attributes in the named node map  
  Calculate att as attlist.$item(k)  
  ; $item() returns the specified attribute in the list  
  Calculate nodetext as con(nodetext,' ',att.$attname,' =  
    ',att.$attvalue)  
  ; $attname and $attvalue are properties of an attribute  
End For  
Quit method nodetext
```

Adding children to a node

Since XML documents are highly structured it is relatively easy to step through the node tree and enumerate all its nodes and sub-nodes (children and grandchildren). You can use the \$childnodes() method to get a list of children for a node, and then construct each child node according to its type by querying its \$nodetype property.

- ❑ **\$childnodes(&cErrorText)**
returns a node list object listing the children of this object, or NULL and cErrorText if an error occurs
- ❑ **\$haschildnodes()**
returns true if the object has children; note no parameters

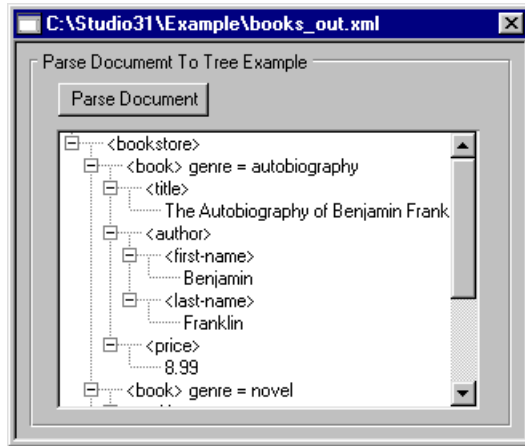
The following method steps through the node tree passed to it and adds the text value of each node to a tree list. Note the switch statement branches on the \$nodetype of the current node and constructs the nodetext accordingly.

```

; Method '$addchildren'
; create parameter vars pObj (Object) and pTree (Item ref)
; create local vars att (Object), attlist (Object), child (Object),
  j (Long int), k (Long int), nl (Object), nodetext (Char),
  treenode (Item ref)
Calculate nl as pObj.$childnodes()
For j from 0 to nl.$length-1 step 1
  Calculate child as nl.$item(j)
  Switch child.$nodetype
    Case kXMLNodeComment
      Calculate nodetext as con('// ',child.$textdata)
    Case kXMLNodeElement
      Do method $getelementtext (child) Returns nodetext
    Case kXMLNodeProcessingInstruction
      Calculate nodetext as con('PI: ',child.$pitarget,
        ' = ',child.$pidata)
    Case kXMLNodeText
      Calculate nodetext as con(child.$textdata)
    Case kXMLNodeCDATASection
      Calculate nodetext as con('CDATA: ',child.$textdata)
    Case kXMLNodeAttribute
      Calculate nodetext as con(child.$attname,
        ' = ',child.$attvalue)
    Case kXMLNodeEntityReference
      Calculate nodetext as 'ER'
    Default
      Calculate nodetext as con('Unexpected node type:
        ',child.$nodetype)
  End Switch
  Set reference treenode to pTree.$add(nodetext)
  If child.$haschildnodes()
    Do method $addchildren (child,treenode)
  End If
End For

```

The above code is available in the example library that accompanies oXML. If you load the sample XML document books.xml you will see something like this:



Saving a document

You can save an XML document to a file on disk using the \$savefile() method or to a binary variable using \$savebinary().

- ❑ **\$savefile**(cPath,&cErrorText[,bStripDT=kFalse,iFmt=kXMLformatNone])
saves XML to pathname cPath; returns true for success, or false and cErrorText; strips prolog DOCTYPE if bStripDT is true; kXMLFormat... constant iFmt controls formatting
- ❑ **\$savebinary**(&xXML,&cErrorText[,bStripDT=kFalse,iFmt=kXMLformatNone])
saves XML to binary variable xXML; returns true for success, or false and cErrorText; strips prolog DOCTYPE if bStripDT is true; kXMLFormat... constant iFmt controls formatting

```

; Method for Save button
; create vars path (Char), errorText (Char), error (Long Int)
On evClick      ;; Event Parameters - pRow( Itemreference )
  Calculate path as
  Do FileOps.$putfilename(path,"Specify name of output XML file")
  If path<>' '
    Do xml.$savefile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    End If
  End If
End If

```

The format parameter for the `$savefile()` and `$savebinary()` methods is an integer and can take one of a number of constants, as follows:

- ❑ **kXMLFormatNone**
The output XML is not formatted; the default if `iFmt` is omitted (that is, no tabs and carriage-return linefeed sequences are inserted)
- ❑ **kXMLFormatBasic**
The output XML is formatted by the insertion of tabs and carriage-return linefeed sequences
- ❑ **kXMLFormatFull**
The output XML is formatted by the insertion of tabs and carriage-return linefeed sequences; in addition, text nodes are formatted by removing all leading and trailing spaces, as well as tabs, carriage returns and linefeeds

Using Lists with XML

You can pass the contents of a document object into an Omnis list variable using the `$savelist()` method. Conversely, you can transfer the contents of an Omnis list, assuming it is in the correct format, to a document object using the `$loadlist()` method. You can therefore manipulate the contents of an XML document via an Omnis list.

- ❑ **\$savelist(&lList,&cErrorText [,bSkipWhiteSpace])**
saves the XML specified by the object into the list `lList`, skipping the whitespace within elements if `bSkipWhiteSpace` is true; returns true for success, or false and `cErrorText` for failure.
- ❑ **\$loadlist(lList,&cErrorText [,cDocTemplatePath])**
loads list `lList` defining an XML document into the object, returns true for success, or false and `cErrorText` for failure. `cDocTemplatePath` is optional and can specify a DTD document template.

The following methods show how you can read an XML document into and out of an Omnis list. The Save List method prompts the user for an XML document, loads the file into the document object, and saves the contents of the object into the Omnis list `xmllist`.

```
; method for save list button
On evClick
  Calculate path as
  Do FileOps.$getfilename(path,"Select XML file to parse")
  If path<>' '
    Calculate xml.$parservalidates as iValidate
    Calculate xml.$replaceentityreferences as iEnt
    Do xml.$loadfile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    Else
      Do xml.$savelist(xmllist,errorText,iSkip) Returns error
    End If
  End If
```

The Load List method prompts the user for a file name for the output XML document using the FileOps method \$putfilename(), prompts the user to identify a DTD for validation, transfers the contents of the Omnis list to the document object, and saves the contents of the document object to the XML disk file.

```
; method for load list button
On evClick
  Calculate path as ''
  Do FileOps.$putfilename(path,"Specify name of output XML file")
  If path<>' '
    Calculate template as
    Do FileOps.$getfilename(template,"Select XML
      document template for output")
    Do xml.$loadlist(xmllist,errorText,template) Returns error
    If error=0
      OK message {Load list failed: [errorText]}
      Quit method
    End If
    Do xml.$savefile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    End If
  End If
```

Using Tree lists with XML

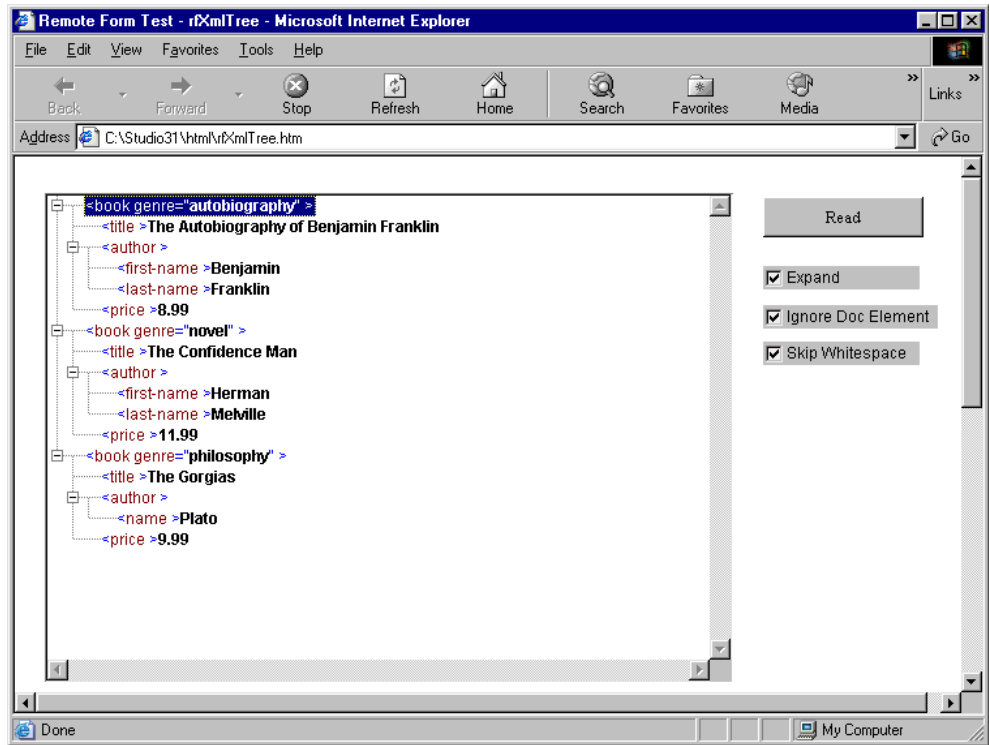
You can save or transfer a document object containing an XML document to an Omnis list variable and display the document in a tree list object using the \$savetree() method. The

tree list can be either a standard window tree control or a web component displayed on a remote form.

- ❑ **\$savetree**(&IList, &cErrorText, bExpanded [,bIgnoreDocumentElement, bSkipWhiteSpace])
 saves the XML specified by the document object to a list suitable for displaying in a data bound tree list object with the dataname IList, and returns true for success, or false and cErrorText for failure;

If true, bExpanded specifies that the tree list is expanded when displayed, bIgnoreDocumentElement specifies that the document root is ignored and not displayed, and bSkipWhiteSpace specifies whether or not white spaces within elements are ignored.

The \$savetree() method can be used behind a button or FormFile object to transfer an XML document into a tree list for display. For example, the following remote form allows the user to view an XML document in a web browser.



The following method is behind a Read button (FormFile object) on the remote form. The method prompts the user to locate an XML document and displays the file in a web tree.

```

; create vars iXml (Object), iList (List)
; iExpanded, iIgnoreDocElem, iSkipWhitespace are linked to the check
; boxes on the form
; pFileData is passed to the method from the FormFile component and
; in this case contains the data from the XML document
On evFileRead
    Calculate iXml.$parservalidates as kFalse
    Do iXml.$loadbinary(cErrorText,cSearchDir) Returns #1
    If #1=0
        Do $cinst.$showmessage(cErrorText)
    Else
        Do iXml.$savetree(iList, cErrorText, iExpanded,
            iIgnoreDocElem, iSkipWhitespace) Returns #1
        If #1=0
            Do $cinst.$showmessage(cErrorText)
        End If
        Do $cinst.$redraw()
    End If

```

The first parameter of the \$savetree() method is an Omnis list variable. When the method is executed the list is populated with the XML data from the document object. The list has a number of columns that are required to draw the tree list. The columns are: nodeType, path, value, attributes, iconid, ident, canedit, flags, and textcolor which are required to draw the tree list.

	type	path	value	attributes	iconid	ident	canedit
1	3004	/bookstore		(Not empty)	0	1	False
2	3004	/bookstore/book		(Not empty)	0	2	False
3	3004	/bookstore/book/title	The Autobiography of Benjamin Franklin	(Not empty)	0	3	False
4	3004	/bookstore/book/author		(Not empty)	0	4	False
5	3004	/bookstore/book/author/first-name	Benjamin	(Not empty)	0	5	False
6	3004	/bookstore/book/author/last-name	Franklin	(Not empty)	0	6	False
7	3004	/bookstore/book/price	8.99	(Not empty)	0	7	False
8	3004	/bookstore/book		(Not empty)	0	8	False
9	3004	/bookstore/book/title	The Confidence Man	(Not empty)	0	9	False

Document Templates

If you wish to build a document containing an XML construct such as a DTD or something that cannot be built using oXML, then provided that this information is fixed for each XML document, you can handle this by using a document template. The approach to building a document becomes:

1. Load the document template; this may contain an inline DTD or link to external DTD file, for example.
2. Add information to the document (elements, text etc.).
3. Save the document.

Character Sets and Unicode

XML documents can contain characters from any language including those represented by Unicode. oXML only works with documents that contain characters that can be converted to the local code page of the environment in which Studio is running, for example, under Windows the ANSI character set is used. Documents containing other characters can be loaded, but will not have the correct data when used in Omnis Studio.

Creating XML documents

The oXML external component makes the handling of XML documents simple within an Omnis application. It treats each node as a separate object, enabling easy searching and manipulation of these nodes within the document. To create an XML document you need to create an Omnis Object with the subtype DOM document object and add different elements to the object. You can do this using the methods built into the document object.

Creating an XML Document

To create an XML document, first you must create a DOM document object (in the method editor Variable pane, Type is Object, and Subtype is under external objects -> DOM document, as described earlier in this manual). This is the master object of your document, and allows you to create, search, edit and delete all types of nodes within your document.

Adding an Element

Your XML document will be made up of several 'elements'. Each element must have a name, but may also have several other properties associated with it, such as attributes, text, and comments. These will be discussed later.

Each element may also contain other elements (known as its children), thereby creating the tree type structure associated with the XML document.

Elements (and all other objects) are created by the DOM document object using its `$createXXX()` methods. For example, the following method returns an element object, `oRoot`, named 'Root':

```
; Define oXML as Object, with subtype DOM document object
; Define lError as Character
; Define oRoot as Object, with subtype DOM Element object
; Define oObj as Object, no subtype
Do oXML.$createelement('Root',lError) returns oRoot
```

Once you have created an element object, you must insert it into the document. This must be done from the element object that will become the parent of the element you are about to add. The element object has two methods for this:

- ❑ `$appendchild()`
inserts the element at the end of its list of children.
- ❑ `$insertbefore()`
inserts the element before the stated object.

As there are no elements when you create your first element, you must use the DOM document object as its parent: this creates what is known as the 'Root' element, of which there can only be one, and all other elements are descendants of this.

```
Do oXML.$appendchild(oRoot ,lError)
Do oXML.$createelement('Element1' ,lError) returns oObj
Do oRoot.$appendchild(oObj, lError)
```

The above method inserts the Root element into the document, then creates another element (Element1), which it returns in `oObj` (since `oObj` has no subtype, its type is defined when it has an object assigned to it; this keeps the number of variables down). The element is then inserted as a child of `oRoot`.

Properties of Elements

Although you have now added some elements, they contain no information. An element may have various properties associated with it. These are all added as children of the element, in the same way that elements are added as children of their parent elements. These may be added before or after inserting the element.

Adding Text

There are two possible ways of displaying text, parsed and unparsed. The usual method is parsed, which means your XML parser will evaluate the text. For example, "Apples & Pears" will be equated to "Apples & Pears". Using unparsed will not evaluate the text and so will express it literally, in this case, "Apples & Pears".

```
; to create PARSED text
Do oXML.$createtextnode(
    "Apples & Pears", lError) returns oObj
; to create UNPARSED text
Do oXML.$createcdatasection(
    "Apples & Pears", lError) returns oObj
```

Will create a text node containing the text, then to add it to the element oElement:

```
Do oElement.$appendChild(oObj, lError)
```

Adding Attributes

Attributes are added in a very simple manner. They require just a name and a value, and are added to the element with its \$setAttribute() method, as follows:

```
Do oElement.$setAttribute("Colour", "red", lError)
```

This will add the attribute Colour = "red" to the element oElement. You can use this method many times to add multiple attributes to the same element. Attributes can also be added using the usual method, such as oXML.\$createattribute(), then the attribute is added as a child of element.

Adding Comments

Comments are not processed by XML parsers, but are present only to improve readability of the XML document. They follow the general form:

```
Do oXML.$createcomment("Your comment here", lError) Returns oObj
Do oElement.$appendChild(oObj, lError)
```

Processing Instructions

Processing instructions are used in XML as a way to keep processor-specific information in the text of the document. They store a 'target' and a value to pass that target. Again, these are created in the same way as the other objects, that is, oXML.\$createprocessinginstruction(), then the processing instruction is added as a child of an element.

Entities

Entities are declared in the DOM document's DocumentType object. The oXML component is based on DOM level 2, which does not support the editing or creation of DocumentType objects. Therefore, the oXML component only allows the reading of entities already defined in an existing XML document.

Saving the XML File

Once you have created your DOM document in Omnis, with all the elements and so on in place, you need to save the document object to an .xml file. To do this, you can use the \$savefile() method in the DOM document object.

```
Do oXML.$savefile("C:\MyFolder\MyXML.xml", lError, kFalse,  
    kXMLFormatFull)
```

The last argument of the `$savefile()` method allows you to specify the formatting of the output XML file. The formatting options let you specify whether or not to add carriage returns and line feeds, and to remove spaces, etc. Different parsers may require different formatting settings to display your XML file.

Reference

This section documents the properties and methods of XML document objects, such as nodes, elements, attributes, and so on. It also includes functions of the document object.

Parameter Syntax

The method parameters are specified using the following naming conventions according to their data type:

Parameter name	Description
cParamName	Character variable, e.g. cName
iParamName	Integer variable, e.g. iIndex
oParamName	Object variable, e.g. oNewObj
bParamName	Boolean variable, e.g. bRecursive
lParamName	List variable, e.g. lList
xParamName	Binary variable, e.g. xData
&ParamName	the parameter receives a return value, e.g. &cErrorText

Common

Properties

All objects have the following properties. For elements, these properties refer to the node containing the element.

Property	Description
\$localname	The local part of the qualified name of the node (read only)
\$namespaceuri	The namespace URI of the node, or empty if it is unspecified (read only)
\$nodename	The name of the node (not assignable)
\$nodetype	The type of the node (one of the kXMLNode... constants)
\$prefix	The namespace prefix of the node, or empty if it is unspecified

Attribute

The DOM Attribute represents an attribute. There are no methods and the \$attvalue is the only property you can assign.

Properties

Property	Description
\$attname	The name of the attribute.
\$attspecified	If true, the attribute was explicitly given a value.
\$attvalue	The value of the attribute.
\$element	The DOM element object which owns the attribute, or NULL if the attribute is not in use (read only property).

CDATASection

The DOM CDATA Section represents the text or content of the CDATA Section of an attribute definition.

Properties

Property	Description
\$textdata	The text or content of the CDATA Section.

Methods

Method	Description
\$splittext()	\$splittext(iOffset,&cErrorText) splits the text at the zero-based offset iOffset; returns the new text object for success, or NULL and cErrorText for failure.

Comment

The DOM Comment represents the content of a comment.

Properties

Property	Description
\$textdata	The text or content of the comment.

Document

The DOM Document object represents an XML document. Its methods allow you to manipulate the XML document and create and retrieve elements and attributes. The document functions are also included here.

Properties

The DOM Document has the following properties, as well as the common properties.

Property	Description
\$ignoreparserwarnings	If true, the parser ignores problems categorized as warnings
\$includeignorablewhitespace	If true, and \$parservalidates is also true, the parser includes ignorable whitespace in the DOM tree it generates
\$outputencoding	The output encoding used by \$savefile and \$savebinary; a constant: kXMLEncodingISO8859 kXMLEncodingUSASCII kXMLEncodingUTF16BE kXMLEncodingUTF16LE kXMLEncodingUTF8
\$parservalidates	If true, the document must have a DTD or schema against which the parser validates the document during \$loadfile and \$loadbinary
\$replaceentityreferences	If true, the parser replaces each entity reference with the value to which it resolves

Methods

Method	Description
\$appendchild	\$appendchild(oObj, &cErrorText) appends oObj to the end of the list of children; returns true for success, or false and cErrorText for failure
\$childnodes	\$childnodes(&cErrorText) returns a node list object listing the children of this object, or NULL and cErrorText if an error occurs
\$clonenode	\$clonenode(bRecursive) returns a new object which is a copy of the node (and its children if bRecursive is true)(note: cloning an element with bRecursive = false also clones the attributes)
\$createattribute	\$createattribute(cName, &cErrorText) returns a new DOM Attr object with the specified attribute name, or NULL and cErrorText if an error occurs
\$createattributens	\$createattributeNS(cURI, cQualifiedName, &cErrorText) returns a new DOM Attr object with the specified URI and name, or NULL and cErrorText if an error occurs
\$createcdatasection	\$createcdatasection(cData, &cErrorText) returns a new DOM CDATA Section object containing the string cData, or NULL and cErrorText if an error occurs
\$createcomment	\$createcomment(cData, &cErrorText) returns a new DOM Comment object containing the string cData, or NULL and cErrorText if an error occurs
\$createdocumentfragment	\$createdocumentfragment(&cErrorText) returns a new empty DOM Document Fragment object, or NULL and cErrorText if an error occurs
\$createelement	\$createelement(cTagName, &cErrorText) returns a new DOM Element object with the specified tag name, or NULL and cErrorText if an error occurs
\$createelementns	\$createelementNS(cURI, cQualifiedName, &cErrorText) returns a new DOM Element object with the specified URI and name, or NULL and cErrorText if an error occurs
\$createentityreference	\$createentityreference(cName, &cErrorText) returns a new DOM Entity Reference object with the specified attribute name, or NULL and cErrorText if an error occurs
\$createprocessing instruction	\$createprocessinginstruction(cTarget, cData, &cErrorText) returns a new DOM Processing Instruction object containing the target cTarget and string cData, or NULL and

Method	Description
	cErrorText if an error occurs
\$createtextnode	\$createtextnode(cData, &cErrorText) returns a new DOM Text object containing the string cData, or NULL and cErrorText if an error occurs
\$doctype	\$doctype() returns the DOM Document Type object for this XML document
\$documentelement	\$documentelement() returns the DOM Element object representing the root of this XML document
\$firstchild	\$firstchild() returns the first child of this object; NULL if there are no children
\$getchildbyid	\$getchildbyid(iChildId) returns the object corresponding to the non-zero child id iChildId, or null if no such child exists
\$getelementbyid	\$getelementbyid(cId, &cErrorText) returns the DOM Element object with the specified id, or NULL and cErrorText if an error occurs
\$getelementsbytagname	\$getelementsbytagname(cTagName, &cErrorText) returns a node list object listing the elements with the name cTagName, or NULL and cErrorText if an error occurs. cTagName = '*' matches all tag names
\$getelementsbytagnameNS	\$getelementsbytagnameNS(cURI, cLocalName, &cErrorText) returns a node list object listing the elements with the URI cURI and name cLocalName, or NULL and cErrorText if an error occurs. '*' matches all URIs or local names or both
\$hasattributes	\$hasattributes() returns true if the node is an element which has attributes
\$haschildNodes	\$haschildNodes() returns true if the object has children
\$importnode	\$importnode(oNode, bDeep, &cErrorText) creates an object containing a copy of the node oNode, copied recursively if bDeep is true; returns the object, or NULL and cErrorText if an error occurs
\$insertbefore	\$insertbefore(oObjToInsert, oObjBefore, &cErrorText) inserts oObjToInsert into the list of children, before oObjBefore; returns true for success, or false and cErrorText for failure
\$issupported	\$issupported(cFeature, cVersion) returns true if the specified version of the specified feature is supported
\$lastchild	\$lastchild() returns the last child of this object; NULL if

Method	Description
	there are no children
\$loadbinary	\$loadbinary(xData, &cErrorText [,cSearchDir]) loads the binary XML data stream xData into the document object; returns true for success, or false and cErrorText for failure
\$loadfile	\$loadfile(cPath, &cErrorText) loads the XML file with pathname cPath into the document object; returns true for success, or false and cErrorText for failure
\$loadlist	\$loadlist(IList, &cErrorText, [cDocTemplatePath]) loads list IList defining an XML document into object; returns true for success, or false and cErrorText for failure.cDocTemplatePath is an empty XML document; this allows a DTD to be specified
\$nextsibling	\$nextsibling() returns the next sibling of this object; NULL if there is no next sibling
\$ownerdocument	\$ownerdocument() returns the owner document containing this object
\$parentnode	\$parentnode() returns the parent object of this object; NULL if there is no parent
\$previousibling	\$previousibling() returns the previous sibling of this object; NULL if there is no previous sibling
\$removechild	\$removechild(oObj, &cErrorText) removes oObj from the list of children; returns true for success, or false and cErrorText for failure
\$replacechild	\$replacechild(oObjNew, oObjOld, &cErrorText) replaces oObjOld in the list of children, with oObjNew; returns true for success, or false and cErrorText for failure
\$savebinary	\$savebinary(&xXML, &cErrorText [,bStripDT=kFalse, iFmt=kXMLformatNone]) saves XML to binary variable xXML;returns true for success,or false and cErrorText;strips prolog DOCTYPE if bStripDT is true;kXMLFormat... constant iFmt controls formatting
\$savefile	\$savefile(cPath, &cErrorText [,bStripDT=kFalse, iFmt=kXMLformatNone]) saves XML to pathname cPath; returns true for success, or false and cErrorText; strips prolog DOCTYPE if bStripDT is true; kXMLFormat... constant iFmt controls formatting
\$savelist	\$savelist(&IList, &cErrorText [,bSkipWhiteSpace=kTrue]) saves the XML specified by the object into the list IList,

Method	Description
	skipping whitespace if specified; returns true for success, or false and cErrorText for failure
\$savetree	\$savetree(&lList, &cErrorText, bExpanded [,bIgnoreDocumentElement=kFalse, bSkipWhiteSpace=kTrue]) saves the XML specified by the document object to a list suitable for displaying in a data bound tree object
\$setempty	\$setempty() discards the current contents of the document object; returns true for success, false for failure

Additional \$qck methods

The following methods allow faster building of XML, without the need to create the intermediate objects, and use integer ids to identify nodes. Note these methods are methods of a DOM Document object but they are not part of the DOM. They are all prefixed by “\$qck” for easy identification.

Method	Description
\$qckappendcdatasection()	\$qckappendcdatasection(iChildId, cData, &cErrorText) appends new CDATA Section node to end of list of children of node iChildId; returns new node id for success, or zero and cErrorText for failure.
\$qckappendcomment()	\$qckappendcomment(iChildId,cData,&cErrorText) appends new comment node to end of list of children of node iChildId; returns new node id for success, or zero and cErrorText for failure.
\$qckappendelement()	\$qckappendelement(iChildId, cTagName, &cErrorText [,cAttName,cAttValue,...]) appends new element node and up to 4 attributes to end of list children of node iChildId; returns new node id for success, or zero and cErrorText for failure.
\$qckappendprocessinginstruction()	\$qckappendprocessinginstruction(iChildId, cTarget, cData, &cErrorText) appends new processing instruction node to end of list of children of node iChildId; returns new node id for success, or false and cErrorText for failure.
\$qckappendtext()	\$qckappendtext(iChildId, cData, &cErrorText) appends new text node to end of list of children of node iChildId; returns new node id for success, or zero and cErrorText for failure.

Document Fragment

The DOM Document Fragment object can represent a fragment or piece of an XML document which can be inserted into a Document. In addition to the common properties, a Document Fragment has the following methods.

Methods

Method	Description
\$appendchild	\$appendchild(oObj, &cErrorText) appends oObj to the end of the list of children; returns true for success, or false and cErrorText for failure
\$childnodes	\$childnodes(&cErrorText) returns a node list object listing the children of this object, or NULL and cErrorText if an error occurs
\$clonenode	\$clonenode(bRecursive) returns a new object which is a copy of the node (and its children if bRecursive is true)(note: cloning an element with bRecursive = false also clones the attributes)
\$firstchild	\$firstchild() returns the first child of this object; NULL if there are no children
\$hasattributes	\$hasattributes() returns true if the node is an element which has attributes
\$haschildnodes	\$haschildnodes() returns true if the object has children
\$insertbefore	\$insertbefore(oObjToInsert, oObjBefore, &cErrorText) inserts oObjToInsert into the list of children, before oObjBefore; returns true for success, or false and cErrorText for failure
\$issupported	\$issupported(cFeature, cVersion) returns true if the specified version of the specified feature is supported
\$lastchild	\$lastchild() returns the last child of this object; NULL if there are no children
\$nextsibling	\$nextsibling() returns the next sibling of this object; NULL if there is no next sibling
\$ownerdocument	\$ownerdocument() returns the owner document containing this object
\$parentnode	\$parentnode() returns the parent object of this object; NULL if there is no parent
\$previoussibling	\$previoussibling() returns the previous sibling of this object; NULL if there is no previous sibling
\$removechild	\$removechild(oObj, &cErrorText) removes oObj from the list of children; returns true for success, or false and cErrorText for failure

Method	Description
\$replacechild	\$replacechild(oObjNew, oObjOld, &cErrorText) replaces oObjOld in the list of children, with oObjNew; returns true for success, or false and cErrorText for failure

DocumentType

The DOM Document Type object provides access to the list of entities and notations (unparsed entities or processing instruction definitions) in the document type of the XML document.

Properties

Property	Description
\$doctype	The name of the DTD, i.e. the name specified in the DOCTYPE element.
\$internalsubset	The internal subset as a string (read only property).
\$publicid	The public identifier of the external subset (read only property).
\$systemid	The system identifier of the external subset (read only property).

Methods

Method	Description
\$entities()	\$entities(&cErrorText) returns a named node map object listing the entities in the document type, or NULL and cErrorText if an error occurs.
\$notations()	\$notations(&cErrorText) returns a named node map object listing the notations in the document type, or NULL and cErrorText if an error occurs.

Element

The DOM Element object can represent any element within a document, including the root element, and provides access to an element's attributes and their values, if the element has any.

Properties

Property	Description
\$tagName	The name of the tag for this element.

Methods

Method	Description
\$attributemap()	\$attributemap(&cErrorText) returns a named node map object listing the attributes of this element, or NULL and cErrorText if an error occurs; the named node map contains an unordered list of attributes
\$getattribute()	\$getattribute(cName,&cErrorText) returns the value of the attribute with name cName; if an error occurs, cErrorText is not empty.
\$getattributenode()	\$getattributenode(cName,&cErrorText) returns the attribute object/node for the attribute with name cName; returns NULL and cErrorText if an error occurs.
\$getattributenodens()	\$getattributenodens(cURI,cLocalName,&cErrorText) returns the attribute object for the attribute with the specified local name and URI; returns NULL and cErrorText if an error occurs.
\$getattributens()	\$getattributens(cURI,cLocalName,&cErrorText) returns the value of the attribute with the specified local name and URI; if an error occurs, cErrorText is not empty.
\$getelementsbytagname()	\$getelementsbytagname(cTagName,&cErrorText) returns a node list object listing the descendant elements with the name cTagName, or NULL and cErrorText if an error occurs. cTagName = '*' matches all tag names.
\$getelementsbytagnamens()	\$getelementsbytagnamens(cURI,cLocalName,&cErrorText) returns node list object listing descendant elements with specified local name and URI, or NULL and cErrorText if an error occurs. '*' in the appropriate parameter matches all URIs or local names or both.

Method	Description
\$hasattribute()	\$hasattribute(cName,&cErrorText) returns true if the element has the named attribute, or the named attribute has a default value. Returns false and cErrorText if an error occurs.
\$hasattributens()	\$hasattributens(cURI,cLocalName,&cErrorText) returns true if the element has the attribute specified by URI and name, or the specified attribute has a default value. Returns false and cErrorText if an error occurs.
\$normalize()	\$normalize(&cErrorText) normalizes the text nodes beneath this element; returns true for success, or false and cErrorText for failure. Text objects within an element may be separated by markup, therefore normalization merges any adjacent text objects into a single text node.
\$removeattribute()	\$removeattribute(cName,&cErrorText) removes the attribute cName; returns true for success, or false and cErrorText for failure.
\$removeattributenode()	\$removeattributenode(oAttr,&cErrorText) removes the attribute oAttr; returns true for success, or false and cErrorText for failure.
\$removeattributens()	\$removeattributens(cURI,cLocalName,&cErrorText) removes the attribute with the specified local name and URI; returns true for success, or false and cErrorText for failure.
\$setattribute()	\$setattribute(cName,cValue,&cErrorText) sets the value of the attribute cName to cValue; returns true for success, or false and cErrorText for failure.
\$setattributenode()	\$setattributenode(oAttr,&cErrorText) replaces the attribute node with oAttr; if attribute is not present returns NULL, otherwise returns previous attribute; if an error occurs, cErrorText is not empty.
\$setattributenodens()	\$setattributenodens(oAttr,&cErrorText) replaces the attribute node with oAttr; if attribute is not present returns NULL, otherwise returns previous attribute; if an error occurs, cErrorText is not empty.
\$setattributens()	\$setattributens(cURI,cQualifiedName,cValue,&cErrorText) sets the value of the attribute with qualified name and URI to cValue; returns true for success, or false and cErrorText for failure.

Entity

The DOM Entity object represents both parsed entities (e.g. ISO approved) and unparsed entities (notations).

Properties

Property	Description
\$notation	For unparsed entities (notations), the name of the notation for the entity. For parsed entities, this is empty. Read only
\$publicid	The public identifier associated with the entity, if specified. Read only
\$systemid	The system identifier associated with the entity, if specified. Read only

NamedNodeMap

The DOM NamedNodeMap object represents an unordered list of nodes. The items in the list are accessible via the index or node name.

Properties

Property	Description
\$length	The number of entries in the named node map. Read only

Methods

Method	Description
\$getnameditem()	\$getnameditem(cName) returns the object named cName, or NULL if no such object exists.
\$getnameditemns()	\$getnameditemns(cURI,cLocalName) returns the object with the specified local name and URI, or NULL if no such object exists.
\$item()	\$item(iIndex) returns object iIndex from the map; indexing starts at zero; a bad index results in a NULL return value.
\$removenameditem()	\$removenameditem(cName) removes and returns the object named cName from the map; returns NULL if no such object exists.
\$removenameditemns()	\$removenameditemns(cURI,cLocalName) removes and returns the object with the specified local name and URI from the map; returns NULL if no such object exists.
\$setnameditem()	\$setnameditem(oObj,&cErrorText) stores the object oObject in the map; returns the previous object if an object is replaced, or NULL if not; if an error occurs, cErrorText is not empty.
\$setnameditemns()	\$setnameditemns(oObj,&cErrorText) stores the object oObject in the map; returns the previous object if an object is replaced, or NULL if not; if an error occurs, cErrorText is not empty.

Node

The DOM represents a document as a tree of “nodes”, where each node (or subnode) in the tree has the following properties and methods as well as its own properties and methods depending on its nodetype. In this sense, many of the properties and methods in this section are generic and can be applied to any node in the tree.

Properties

Property	Description
\$localname	The local part of the qualified name of the node (read only property).
\$namespaceuri	The namespace URI of the node, or empty if it is unspecified (read only property).
\$nodename	The name of the node
\$nodetype	The type of the node: one of the constants: <code>kXMLElement</code> , <code>kXMLNodeAttribute</code> , <code>kXMLNodeText</code> , <code>kXMLNodeCDATASection</code> , <code>kXMLNodeCDATASection</code> , <code>kXMLNodeEntityReference</code> , <code>kXMLNodeEntity</code> , <code>kXMLNodeProcessingInstruction</code> , <code>kXMLNodeComment</code> , <code>kXMLNodeDocument</code> , <code>kXMLNodeDocumentType</code> , <code>kXMLNodeDocumentFragment</code> , <code>kXMLNodeNotation</code>
\$prefix	The namespace prefix of the node, or empty if it is unspecified.

Methods

Method	Description
\$appendchild()	\$appendchild(oObj,&cErrorText) appends oObj to the end of the list of children; returns true for success, or false and cErrorText for failure.
\$childnodes()	\$childnodes(&cErrorText) returns a node list object listing the children of this object, or NULL and cErrorText if an error occurs.
\$clonenode()	\$clonenode(bRecursive) returns a new object which is a copy of the node (and its children if bRecursive is true); note cloning an element with bRecursive = false also clones the attributes.
\$firstchild()	Returns the first child of this object; NULL if there are no children.
\$hasattributes()	Returns true if the node is an element which has attributes.
\$haschildnodes()	Returns true if the object has children.
\$insertbefore()	\$insertbefore(oObjToInsert,oObjBefore,&cErrorText) inserts oObjToInsert into the list of children, before oObjBefore; returns true for success, or false and cErrorText for failure.
\$issupported()	\$issupported(cFeature,cVersion) returns true if the specified version of the specified DOM feature is supported.
\$lastchild()	Returns the last child of this object; NULL if there are no children.
\$nextsibling()	Returns the next sibling of this object; NULL if there is no next sibling.
\$ownerdocument()	Returns the owner document containing the object.
\$parentnode()	Returns the parent object of this object; NULL if there is no parent.
\$previous sibling()	Returns the previous sibling of this object; NULL if there is no previous sibling.
\$removechild()	\$removechild(oObj,&cErrorText) removes oObj from the list of children; returns true for success, or false and cErrorText for failure.
\$replacechild()	\$replacechild(oObjNew,oObjOld,&cErrorText) replaces oObjOld in the list of children, with oObjNew; returns true for success, or false and cErrorText for failure.

NodeList

The DOM NodeList object represents an ordered list of nodes. The items in the node list are accessible via an integral index which starts at 0 for the first node.

Properties

Property	Description
\$length	The number of nodes in the node list. Read only

Methods

Method	Description
\$item()	\$item(iIndex) returns the node from the node list specified by iIndex, where indexing starts at zero; a bad index results in a NULL return value.

Notation

The DOM Notation object represents unparsed entities (as opposed to parsed/standard ones), or this object could refer to the formal declaration of a Processing Instruction target.

Properties

Property	Description
\$publicid	The public identifier of the notation. Read only
\$systemid	The system identifier of the notation. Read only

ProcessingInstruction

The DOM Processing Instruction object is a special element or tag that provides instructions parsed to an external application, e.g. the XML version declaration at the beginning of a document is a processing instruction.

Properties

Property	Description
\$pdata	The data or contents of the processing instruction.
\$target	The target of the processing instruction. The target is defined as the first token in a processing instruction that identifies the application to which the instruction is directed, e.g. “xml” is the target in the XML version declaration of a document. Read only

Text

The DOM Text object represents the text or content of an element or attribute.

Properties

Property	Description
\$textdata	The text or content of the element or attribute.

Methods

Method	Description
\$splittext()	\$splittext(iOffset,&cErrorText) splits the text at the zero-based offset iOffset; returns the new text object for success, or NULL and cErrorText for failure.

Constants

Node Types

DOM represents an XML document as a hierarchy of nodes. The \$nodetype can be one of the following values.

Constant	Description
kXMLElement	The node is an Element
kXMLNodeAttribute	The node is an Attribute
kXMLNodeText	The node is a Text node
kXMLNodeCDATASection	The node is a CDATASection
kXMLNodeEntityReference	The node is an EntityReference
kXMLNodeEntity	The node is an Entity
kXMLNodeProcessingInstruction	The node is a ProcessingInstruction
kXMLNodeComment	The node is a Comment
kXMLNodeDocument	The node is a Document
kXMLNodeDocumentType	The node is a DocumentType
kXMLNodeDocumentFragment	The node is a DocumentFragment
kXMLNodeNotation	The node is a Notation

Encoding Types

The \$outputencoding property used by \$savefile() and \$savebinary() methods can be one of the following values.

Constant	Description
kXMLEncodingISO8859	ISO8859 encoding
kXMLEncodingUSASCII	USASCII encoding
kXMLEncodingUTF16BE	UTF16BE encoding
kXMLEncodingUTF16LE	UTF16LE encoding
kXMLEncodingUTF8	UTF8 encoding

Functions

The following functions are available in the oXML component.

OXML.\$base64decode()

Function group	Execute on Web Client	Platform(s)
OXML	NO	All

Syntax

OXML.\$base64decode(*cData*,&*cErrorText*)

Description

Decodes the BASE64 encoded *data* and returns the result, or NULL and *cErrorText* if an error occurs.

OXML.\$base64encode()

Function group	Execute on Web Client	Platform(s)
OXML	NO	All

Syntax

OXML.\$base64encode(*xData*,&*cErrorText*)

Description

Encodes the *data* using BASE64 and returns the result, or NULL and *cErrorText* if an error occurs.

OXML.\$formatbinaschar()

Function group	Execute on Web Client	Platform(s)
OXML	NO	All

Syntax

OXML.\$formatbinaschar(*xData*[,*iBytesPerRow*=16,*iMaxLength*=0])

Description

Formats at most the first *iMaxLength* bytes of binary data *xData* into character data suitable for display in a multi-line entry field.

OXML.\$maybexml()

Function group	Execute on Web Client	Platform(s)
OXML	NO	All

Syntax

OXML.\$maybeXML(*xData*)

Description

Returns true if the binary *data* is likely to be XML (it starts with <?xml, in one of the encodings supported by the parser).

OXML.\$striphttpheader()

Function group	Execute on Web Client	Platform(s)
OXML	NO	All

Syntax

OXML.\$striphttpheader(&*cData*)

Description

Strips the HTTP header from the *cData* argument.

Index

- \$appendchild, 36, 40
- \$appendchild(), 30, 47
- \$attname, 34
- \$attributemap(), 21, 42
- \$attspecified, 34
- \$attvalue, 34
- \$base64decode(), 51
- \$base64encode(), 51
- \$childnodes, 36, 40
- \$childnodes(), 22, 47
- \$clonenode, 36, 40
- \$clonenode(), 47
- \$createattribute, 36
- \$createattributens, 36
- \$createdatasection, 36
- \$createcomment, 36
- \$createdocumentfragment, 36
- \$createelement, 36
- \$createelementns, 36
- \$createentityreference, 36
- \$createprocessinginstruction, 36
- \$createtextnode, 37
- \$doctype, 37
- \$documentelement, 37
- \$documentelement(), 21
- \$dtdname, 41
- \$element, 34
- \$entities(), 41
- \$firstchild, 37, 40
- \$firstchild(), 47
- \$formatbinaschar(), 51
- \$getattribute(), 42
- \$getattributenode(), 42
- \$getattributenodens(), 42
- \$getattributens(), 42
- \$getchildbyid, 37
- \$getelementbyid, 37
- \$getelementsbytagname, 37
- \$getelementsbytagname(), 42
- \$getelementsbytagnamens, 37
- \$getelementsbytagnamens(), 42
- \$getnameditem(), 45
- \$getnameditemns(), 45
- \$hasattribute(), 43
- \$hasattributens(), 43
- \$hasattributes, 37, 40
- \$hasattributes(), 47
- \$haschildnodes, 37, 40
- \$haschildnodes(), 22, 47
- \$ignoreparserwarnings, 35
- \$importnode, 37
- \$includeignorablewhitespace, 35
- \$insertbefore, 37, 40
- \$insertbefore(), 30, 47
- \$internalsubset, 41
- \$issupported, 37, 40
- \$issupported(), 47
- \$item(), 21, 45, 48
- \$lastchild, 37, 40
- \$lastchild(), 47
- \$length, 45, 48
- \$loadbinary, 38
- \$loadbinary(), 19
- \$loadfile, 38
- \$loadfile(), 19
- \$loadlist, 38
- \$loadlist(), 25
- \$localname, 33, 46
- \$maybexml(), 52
- \$namespaceuri, 33, 46
- \$nextsibling, 38, 40
- \$nextsibling(), 47
- \$nodename, 33, 46
- \$nodetype, 22, 33, 46
- \$normalize(), 43
- \$notation, 44
- \$notations(), 41
- \$outputencoding, 35
- \$ownerdocument, 38, 40
- \$ownerdocument(), 47
- \$parentnode, 38, 40
- \$parentnode(), 47
- \$parservalidates, 35

- \$pidata, 49
- \$pitable, 49
- \$prefix, 33, 46
- \$previous sibling, 38, 40
- \$previous sibling(), 47
- \$publicid, 41, 44, 48
- \$qckappenddatasection(), 39
- \$qckappendcomment(), 39
- \$qckappendelement(), 39
- \$qckappendprocessinginstruction(), 39
- \$qckappendtext(), 39
- \$removeattribute(), 43
- \$removeattributenode(), 43
- \$removeattributens(), 43
- \$removechild, 38, 40
- \$removechild(), 47
- \$removenameditem(), 45
- \$removenameditemns(), 45
- \$replacechild, 38, 41
- \$replacechild(), 47
- \$replaceentityreferences, 35
- \$savebinary, 38
- \$savebinary(), 24, 25, 50
- \$savefile, 38
- \$savefile(), 24, 25, 31, 50
- \$savelist, 38
- \$savelist(), 25
- \$savetree, 39
- \$savetree(), 27
- \$setattribute(), 43
- \$setattributenode(), 43
- \$setattributenodens(), 43
- \$setattributens(), 43
- \$setempty, 39
- \$setnameditem(), 45
- \$setnameditemns(), 45
- \$splittext(), 34, 49
- \$striphttpheader(), 52
- \$systemid, 41, 44, 48
- \$tagname, 21, 42
- \$textdata, 34, 35, 49

- Attributes, 8, 21

- Change Serial Number, 15
- Children, Adding to a node, 22
- Constants, 50

- Deployment licensing, 15

- Document
 - Root element, 21
- Document objects, 18
 - Creating a Document Object, 16
 - Creating XML objects, 16
- Documents
 - Loading an XML document, 19
 - Saving, 24
- Documents, DOM, 13
- DOM, 12
 - DOM level 2, 6
 - DOM Level 2, 12
 - What is the DOM?, 12
- DOM Attribute, 34
 - Properties, 34
- DOM CDATASection
 - Properties and methods, 34
- DOM Comment
 - Properties, 35
- DOM Document
 - Properties and methods, 35
- DOM DocumentFragment, 40
- DOM DocumentType
 - Properties and methods, 41
- DOM Element
 - Properties and methods, 42
- DOM Entity
 - Properties, 44
- DOM NamedNodeMap
 - Properties and methods, 45
- DOM Node
 - Properties and methods, 46
- DOM NodeList
 - Properties and methods, 48
- DOM Notation
 - Properties, 48
- DOM ProcessingInstruction
 - Properties, 49
- DOM Text
 - Properties and methods, 49
- DTDs, 8

- Elements, 6, 21
- Encoding types, Constants, 50
- Entities, 8

- Functions, 51

- Installation, 14

- kXMLFormatBasic, 25
- kXMLFormatFull, 25
- kXMLFormatNone, 25

- Licensing
 - Deployment, 15
- Lists, with XML, 15

- namespaces*, 9
- Node types, Constants, 50
- Nodes
 - Adding children, 22
- Nodes, DOM, 13
- Notations, 8

- Object variables, 16
- Onduit
 - Serialization, 15

- Parsed entities, 8
- Parsers, 10

- Root element, 21
- Runtime
 - Licensing, 15

- Saving a document, 24
- Schemas, 9
- Serial.txt file, 15
- Serialization, 15
- Server
 - Licensing, 15

- Templates
 - DTDs, 8
 - Schemas, 9
- Tree lists, with XML, 26

- Unparsed entities, 8

- xerces-c_1_6_0.dll, 14
- XML
 - Benefits, 11
 - Introduction, 6
 - Parsers, 10
 - What is XML?, 6
- XML documents
 - Loading, 19