Oracle®

XML API Reference

10*g* Release 1 (10.1)

Part No. B10789-01

December 2003

ORACLE®

Oracle XML API Reference, 10*g* Release 1 (10.1)

Part No. B10789-01

Contributing Authors: Stanley Guan, Dmitry Lenkov, Roza Leyderman, Ian Macky, Anguel Novoselsky, Tomas Saulys, Mark Scardina

Contents

4 Package DOM APIs for C

5 Package Range APIs for C

12 Package XSLT APIs for C

13 Package XSLTVM APIs for C

15 Package Dom APIs for C++

17 Package OracleXml APIs for C++

18 Package Parser APIs for C++

19 Package Tools APIs for C++

20 Package XPath APIs for C++

21 Package XPointer APIs for C++

22 Package Xsl APIs for C++

List of Tables

Send Us Your Comments

Oracle XML API Reference, 10*g* Release 1 (10.1)

Part No. B10789-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:

  Oracle Corporation

  Server Technologies Documentation

  500 Oracle Parkway, Mailstop 4op11

  Redwood Shores, CA 94065 USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This reference describes Oracle XML Developer's Kits (XDK) and Oracle XML DB APIs. It primarily lists the syntax of functions, methods, and procedures associated with these APIs.

This preface contains these topics:

Audience

This guide is intended for developers building XML applications in Oracle.

To use this document, you need a basic understanding of object-oriented programming concepts, familiarity with Structured Query Language (SQL), and working knowledge of application development using either C or C++.

Organization

This document is divided into two parts, for C and C++ language APIs.

Part I, "XML APIs for C"

This part provides information about XML related C APIs. Please note that starting with this release, a new unified group of APIs has been developed to enable consistent development interface both for client-side and server-side applications. The previous C APIs are still available in this release for legacy accommodation, but their use is strongly discouraged. Please note that these separate C APIs will be deprecated in the next Oracle release.

Unified C APIs provided here include:

- Chapter 1, "Mapping Between Old and New C APIs"
- Chapter 2, "Datatypes for C"
- Chapter 3, "Package Callback APIs for C"
- Chapter 4, "Package DOM APIs for C"
- Chapter 5, "Package Range APIs for C"
- Chapter 6, "Package SAX APIs for C"
- Chapter 7, "Package Schema APIs for C"
- Chapter 8, "Package Traversal APIs for C"
- Chapter 9, "Package XML APIs for C"
- Chapter 10, "Package XPath APIs for C"
- Chapter 11, "Package XPointer APIs for C"
- Chapter 12, "Package XSLT APIs for C"
- Chapter 13, "Package XSLTVM APIs for C"

Part II, "XML APIs for C++"

This part provides information about XML related C++ APIs:

- Chapter 14, "Package Ctx APIs for C++"
- Chapter 15, "Package Dom APIs for C++"
- Chapter 16, "Package IO APIs for C++"
- Chapter 17, "Package OracleXml APIs for C++"
- Chapter 18, "Package Parser APIs for C++"
- Chapter 19, "Package Tools APIs for C++"
- Chapter 20, "Package XPath APIs for C++"
- Chapter 21, "Package XPointer APIs for C++"
- Chapter 22, "Package Xsl APIs for C++"

Related Documentation

For more information, see these Oracle resources:

- *Oracle Database Concepts*
- *Oracle Database SQL Reference*
- *Oracle Database Application Developer's Guide - Object-Relational Features*
- *Oracle Database New Features*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership/`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/documentation/`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
. . .	Horizontal ellipsis points indicate either:	
	That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

Convention	Meaning	Example

. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`//process information in buffer`
		`.`
		`.`
		`blob.close();`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME =` *`database_name`*

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The

conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What's New in Oracle XML API Reference?

This section describes new features in *Oracle XML API Reference* and provides pointers to additional information.

10*g* Release 1 (10.1)

The following features are new to this release:

- New APIs for the C programming language; see Part I, "XML APIs for C"
- New APIs for the C++ programming language; see Part II, "XML APIs for C++"
- All APIs for PL/SQL are now documented in *PL/SQL Packages and Types Reference*
- All APIs for SQL are now documented in *Oracle Database SQL Reference*
- All APIs for the Java programming language are now documented in a new book, *Oracle XML Java API Reference*

Part I

XML APIs for C

This part contains the following chapters:

1

Mapping Between Old and New C APIs

This chapter maps the XML C APIs available in Oracle9*i* release to the Unified XML C APIs available in this release of Oracle Database.

The chapter contains these topics:

- C Package Changes
- Initializing and Parsing Sequence Changes
- Datatype Mapping between oraxml and xml Packages
- Method Mapping between oraxml and xml Packages

See Also:

- Format Models in *Oracle XML Developer's Kit Programmer's Guide*

C Package Changes

Pre-existing C APIs were available through the `oraxml` package. It had the following characteristics:

- Specification is limited to a one-to-one mapping between the xml context (`xmlctx`) and an xml document. Only one document can be accessed by DOM at any one time, however the data of multiple documents can be concurrent.
- The APIs are not always consistent, and don't always follow the declarations of the `xmlctx`.

In contrast, the new unified C APIs solve these problems:

- Multiple independent documents share the `xmlctx`.
- All APIs conform to the declarations of the `xmlctx`.
- Each document can be accessed simultaneously by DOM until explicitly destroyed by an `XmlDestroy()` call.

Initializing and Parsing Sequence Changes

The initialization and parsing of documents has changed in the Unified C API.

Example 1–1 Initializing and Parsing Sequence for the Pre-Unified C API, One Document at a Time

The following pseudo-code demonstrates how to initialize and parse documents, one at a time, using the old C APIs. Contrast this with Example 1–2.

```
#include <oraxml.h>
uword err;
xmlctx *ctx = xmlinit(&err, options);
for (;;)
{
   err = xmlparse(ctx, URI, options);
   ...
   /* DOM operations */
   ...
   /* recycle memory from document */
   xmlclean(ctx);
}
xmlterm(ctx);
```

Example 1–2 Initializing and Parsing Sequence for the Unified C API, One Document at a Time

The following pseudo-code demonstrates how to initialize and parse documents, one at a time, using the new C APIs. Contrast this with Example 1–1.

```
#include <xml.h>
xmlerr err;
xmldocnode *doc;
xmlctx *xctx = XmlCreate(&err, options, NULL);
for (;;)
{
   doc = XmlLoadDom(xctx, &err, "URI", URI, NULL);
   ...
   /* DOM operations */
   ...
   XmlFreeDocument(xctx, doc);
}
XmlDestroy(xctx);
```

Example 1–3 Initializing and Parsing Sequence for the Pre-Unified C API, Multiple Documents and Simultaneous DOM Access

The following pseudo-code demonstrates how to initialize and parse multiple documents with simultaneous DOM access using the old C APIs. Contrast this with Example 1–4.

```
xmlctx *ctx1 = xmlinitenc(&err, options);
xmlctx *ctx2 = xmlinitenc(&err, options);
err = xmlparse(ctx1, URI_1, options);
err = xmlparse(ctx2, URI_2, options);
...
/* DOM operations for both documents */
...
xmlterm(ctx1);
xmlterm(ctx2);
```

Example 1–4 Initializing and Parsing Sequence for the Unified C API, Multiple Documents and Simultaneous DOM Access

The following pseudo-code example demonstrates how to initialize and parse multiple documents with simultaneous DOM access using the new C APIs. Contrast this with Example 1–3.

```
xmldocnode *doc1;
xmldocnode *doc2;
xmlctx *xctx = XmlCreate(&err, options, NULL);
doc1 = XmlLoadDom(xctx, &err, "URI", URI_1, NULL);
doc2 = XmlLoadDom(xctx, &err, "URI", URI_2, NULL);
...
/* DOM operations for both documents*/
...
XmlFreeDocument(xctx, doc1);
XmlFreeDocument(xctx, doc2);
...
XmlDestroy(xctx);
```

Datatype Mapping between oraxml and xml Packages

Table 1–1 outlines the changes made to datatypes for the new C API.

Table 1–1 Datatypes Supported by oraxml Package versus xml Package

oraxml Supported Datatype	xml Supported Datatype
uword	xmlerr
xmlacctype	xmlurlacc
xmlattrnode	xmlattrnode
xmlcdatanode	xmlcdatanode
xmlcommentnode	xmlcommentnode
xmlctx	xmlctx
xmldocnode	xmldocnode
xmldomimp	Obsolete. Use xmlctx.
xmldtdnode	xmldtdnode
xmlelemnode	xmlelemnode
xmlentnode	xmlentnode
xmlentrefnode	xmlentrefnode
xmlflags	ub4
xmlfragnode	xmlfragnode
xmlihdl	xmlurlhdl
xmlmemcb	Use individual function pointers.
xmlnode	xmlnode
xmlnodes	xmlnodelist, xmlnamedmap
xmlnotenode	xmlnotenode
xmlntype	xmlnodetype
xmlpflags	ub4
xmlpinode	xmlpinode
xmlsaxcb	xmlsaxcb

Table 1–1 (Cont.) Datatypes Supported by oraxml Package versus xml Package

oraxml Supported Datatype	xml Supported Datatype
xmlstream	xmlistream, xmliostream
xmltextnode	xmltextnode
xpctx	xpctx
xpexpr	xpexpr
xpnset	Obsolete. Use XmlXPathGetObjectNSetNum() and XmlXPathGetObjectNSetNode().
xpnsetele	Obsolete. Use XmlXPathGetObjectNSetNum() and XmlXPathGetObjectNSetNode().
xpobj	xpobj
xpobjtyp	xmlxslobjtype
xslctx	xslctx
xsloutputmethod	xmlxsloutputmethod

Method Mapping between oraxml and xml Packages

Table 1–2 outlines the changes made to the methods of the new C API.

Table 1–2 Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
appendChild()	XmlDomAppendChild()
appendData()	XmlDomAppendData()
cloneNode()	XmlDomCloneNode()
createAttribute()	XmlDomCreateAttr()
createAttributeNS()	XmlDomCreateAttrNS()
createCDATASection()	XmlDomCreateCDATA()
createComment()	XmlDomCreateComment()
createDocument()	XmlCreateDocument()
createDocumentFragment()	XmlDomCreateFragment()
createDocumentNS()	XmlCreateDocument()
createDocumentType()	XmlCreateDTD()
createElement()	XmlDomCreateElem()
createElementNS()	XmlDomCreateElemNS()
createEntityReference()	XmlDomCreateEntityRef()
createProcessingInstruction()	XmlDomCreatePI()
createTextNode()	XmlDomCreateText()
deleteData()	XmlDomDeleteData()
freeElements()	XmlDomFreeNodeList()
getAttribute()	XmlDomGetAttr()
getAttributeIndex()	XmlDomGetAttrs(), XmlDomGetNodeMapItem()
getAttributeNode()	XmlDomGetAttrNode()
getAttributes()	XmlDomGetAttrs()
getAttrLocal()	XmlDomGetAttrLocal(), XmlDomGetAttrLocalLen()

Table 1–2 (Cont.) Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
`getAttrName()`	`XmlDomGetAttrName()`
`getAttrNamespace()`	`XmlDomGetAttrURI(), XmlDomGetAttrURILen()`
`getAttrPrefix()`	`XmlDomGetAttrPrefix()`
`getAttrQualifiedName()`	`XmlDomGetAttrName()`
`getAttrSpecified()`	`XmlDomGetAttrSpecified()`
`getAttrValue()`	`XmlDomGetAttrValue()`
`getCharData()`	`XmlDomGetCharData()`
`getChildNode()`	`XmlDomGetChildNode()`
`getChildNodes()`	`XmlDomGetChildNodes()`
`getContentModel()`	`XmlDomGetContentModel()`
`getDocType()`	`XmlDomGetDTD()`
`getDocTypeEntities()`	`XmlDomGetDTDEntities()`
`getDocTypeName()`	`XmlDomGetDTDName()`
`getDocTypeNotations()`	`XmlDomGetDTDNotations()`
`getDocument()`	Obsolete; document returned by `XmlLoadDom`***`xxx`***`()` calls
`getDocumentElement()`	`XmlDomGetDoctElem()`
`getElementByID()`	`XmlDomGetElemByID()`
`getElementsByTagName()`	`XmlDomGetElemsByTag()`
`getElementsByTagNameNS()`	`XmlDomGetElemsByTag()`
`getEncoding()`	`XmlDomGetEncoding()`
`getEntityNotation()`	`XmlDomGetEntityNotation()`
`getEntityPubID()`	`XmlDomGetEntityPubID()`
`getEntitySysID()`	`XmlDomGetEntitySysID()`
`getFirstChild()`	`XmlDomGetFirstChild()`
`getImplementation()`	Obsolete; use `xmlctx` instead of `DOMImplementation`
`getLastChild()`	`XmlDomGetLastChild()`

Table 1–2 (Cont.) Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
getNamedItem()	XmlDomGetNamedItem()
getNextSibling()	XmlDomGetNextSibling()
getNodeLocal()	XmlDomGetNodeLocal(), XmlDomGetNodeLocalLen()
getNodeMapLength()	XmlDomGetNodeMapLength()
getNodeName()	XmlDomGetNodeName(), XmlDomGetNodeNameLen()
getNodeNameSpace()	XmlDomGetNodeURI(), XmlDomGetNodeURILen()
getNodePrefix()	XmlDomGetNodePrefix()
getNodeQualifiedName()	XmlDomGetNodedName(), XmlDomGetNodedNameLen()
getNodeType()	XmlDomGetNodeType()
getNodeValue()	XmlDomGetNodeValue(), XmlDomGetNodeValueLen()
getNotationPubID()	XmlDomGetNotationPubID()
getNotationSysID()	XmlDomGetNotationSysID()
getOwnerDocument()	XmlDomGetOwnerDocument()
getParentNode()	XmlDomGetParentNode()
getPIData()	XmlDomGetPIData()
getPITarget()	XmlDomGetPITarget()
getPreviousSibling()	XmlDomGetPrevSibling()
getTagName()	XmlDomGetTagName()
hasAttributes()	XmlDomHasAttrs()
hasChildNodes()	XmlDomHasChildNodes()
hasFeature()	XmlHasFeature()
importNode()	XmlDomImportNode()
insertBefore()	XmlDomInsertBefore()
insertData()	XmlDomInsertData()
isSingleChar()	XmlIsSimple()
isStandalone()	XmlDomGetDecl()
isUnicode()	XmlDomIsUnicode()

Table 1–2 (Cont.) Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
nodeValid()	XmlDomValidate()
normalize()	XmlDomNormalize()
numAttributes()	XmlDomNumAttrs()
numChildNodes()	XmlDomNumChildNodes()
prefixToURI()	XmlDomPrefixToURI()
printBuffer()	XmlSaveDomBuffer()
printBufferEnc()	XmlSaveDomBuffer()
printCallback()	XmlSaveDomStream()
printCallbackEnc()	XmlSaveDomStream()
printSize()	XmlSaveDomSize()
printSizeEnc()	XmlSaveDomSize()
printStream()	XmlSaveDomStdio()
printStreamEnc()	XmlSaveDomStdio()
removeAttribute()	XmlDomRemoveAttr()
removeAttributeNode()	XmlDomRemoveAttrNode()
removeChild()	XmlDomRemoveChild()
removeNamedItem()	XmlDomRemoveNamedItem()
replaceChild()	XmlDomReplaceChild()
replaceData()	XmlDomReplaceData()
saveString2()	XmlDomSaveString2()
saveString()	XmlDomSaveString()
setAttribute()	XmlDomSetAttr()
setAttributeNode()	XmlDomSetAttrNode()
setAttrValue()	XmlDomSetAttrValue()
setCharData()	XmlDomSetCharData()
setNamedItem()	XmlDomSetNamedItem()
setNodeValue()	XmlDomSetNodeValue(), XmlDomSetNodeValueLen()

Table 1–2 (Cont.) Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
setPIData()	XmlDomSetPIData()
splitText()	XmlDomSplitText()
substringData()	XmlDomSubstringData()
xmlaccess()	XmlAccess()
xmlinit()	XmlCreate()
xmlinitenc()	XmlCreate()
xmlLocation()	TBD.
xmlparse()	XmlLoadDomURI()
xmlparsebuf()	XmlLoadDomBuffer()
xmlparsedtd()	Obsolete; use XML_LOAD_FLAG_DTD_ONLY flag in XmlLoad*XXX*() calls.
xmlparsefile()	XmlLoadDomFile()
xmlparsestream()	XmlLoadDomStream()
xmlterm()	XmlDestroy()
xmlwhere()	TBD.
xpevalxpathexpr()	XmlXPathEval()
xpfreexpathctx()	XmlXPathDeleteCtx()
xpgetbooleanval()	XmlXPathGetObjectBoolean()
xpgetfirstnsetelem()	XmlXPathGetObjectNSetNum()
xpgetnextnsetelem()	XmlXPathGetObjectNSetNum()
xpgetnsetelemnode()	XmlXPathGetObjectNSetNum()
xpgetnsetval()	XmlXPathGetObjectNSetNum()
xpgetnumval()	XmlXPathGetObjectNumber()
xpgetrtfragval()	XmlXPathGetObjectFragment()
xpgetstrval()	XmlXPathGetObjectString()
xpgetxpobjtyp()	XmlXPathGetObjectType()
xpmakexpathctx()	XmlXPathCreateCtx()

Table 1–2 (Cont.) Methods of the oraxml Package versus the xml Package

Package oraxml Method	Package xml Method(s)
xpparsexpathexpr()	XmlXPathParse()
xslgetbaseuri()	XmlXslGetBaseURI()
xslgetoutputdomctx()	XmlXslGetOutputDom()
xslgetoutputsax()	Unnecessary
xslgetoutputstream()	Unnecessary
xslgetresultdocfrag()	XmlXslGetOutputFragment()
xslgettextparam()	XmlXslGetTextParam()
xslgetxslctx()	Unnecessary
xslinit()	XmlXslCreateCtx()
xslprocess()	XmlXslProcess()
xslprocessex()	XmlXslProcess()
xslprocessxml()	XmlXslProcess()
xslprocessxmldocfrag()	XmlXslProcess()
xslresetallparams()	XmlXslResetAllParams()
xslsetoutputdomctx()	XmlXslSetOutputDom()
xslsetoutputencoding()	XmlXslSetOutputEncoding()
xslsetoutputmethod()	XmlXslSetOutputMethod()
xslsetoutputsax()	XmlXslSetOutputSax()
xslsetoutputsaxctx()	XmlXslSetOutputSax()
xslsetoutputstream()	XmlXslSetOutputStream()
xslsettextparam()	XmlXslSetTextParam()
xslterm()	XmlXslDeleteCtx()

2

Datatypes for C

This package defines macros which declare functions (or function pointers) for XML callbacks. Callbacks are used for error-message handling, memory allocation and freeing, and stream operations.

This chapter contains this section:

- C Datatypes

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

C Datatypes

Table 2–1 lists all C datatypes and their descriptions.

Table 2–1 Summary of C Datatypes

Datatype	Purpose
xmlcmphow on page 2-3	Constant used for DOM Range comparisons.
xmlctx on page 2-3	Context shared for all documents in an XML session.
xmlerr on page 2-4	Numeric error code returned by many functions.
xmlistream on page 2-4	Generic user-defined input stream.
xmliter on page 2-5	Control structure for DOM2 `NodeIterator` and `TreeWalker`.
xmlnodetype on page 2-5	The numeric type code of a node.
xmlostream on page 2-6	Generic user-defined output stream.
xmlpoint on page 2-7	XPointer point location.
xmlrange on page 2-7	Controls structure for DOM2 Range.
xmlshowbits on page 2-8	Bit flags used to select which node types to show.
xmlurlacc on page 2-8	This is an enumeration of the known access methods for retrieving data from a URL.
xmlurlhdl on page 2-9	This union contains the handle(s) needed to access URL data, be it a stream or `stdio` pointer, file descriptor(s), and so on.
xmlurlpart on page 2-9	This structure contains the sub-parts of a URL.
xmlxptrloc on page 2-10	XPointer location datatype.
xmlxptrlocset on page 2-10	XPointer location set datatype.
xmlxslobjtype on page 2-10	Type of XSLT object that may be returned.
xmlxslomethod on page 2-10	Type of output produced by the XSLT processor.
xmlxvm on page 2-11	An object of type `xmlxvm` is used for XML document transformation.
xmlxvmcomp on page 2-11	An object of type `xmlxvmcomp` is used for compiling XSL stylesheets.

Table 2–1 (Cont.) Summary of C Datatypes

Datatype	Purpose
xmlxvmflags on page 2-11	Control flags for the XSLT compiler.
xmlxvmobjtype on page 2-11	Type of XSLTVM object.
xpctx on page 2-12	XPath top-level context.
xpexpr on page 2-12	XPath expression.
xpobj on page 2-12	XPath object.
xsdctx on page 2-12	XMLSchema validator context.
xslctx on page 2-13	XSL top-level context.
xvmobj on page 2-13	XSLVM processor run-time object; contents are private and must not be accessed by users.

xmlcmphow

Constant used for DOM Range comparisons.

Definition

```
typedef enum {
    XMLDOM_START_TO_START = 0,
    XMLDOM_START_TO_END   = 1,
    XMLDOM_END_TO_END     = 2,
    XMLDOM_END_TO_START   = 3
} xmlcmphow;
```

xmlctx

Context shared for all documents in an XML session. Contains encoding information, low-level memory allocation function pointers, error message language/encoding and optional handler function, and so on. Required to load (parse) documents and create DOM, generate SAX, and so on.

Definition

```
struct xmlctx;
typedef struct xmlctx xmlctx;
```

xmlerr

Numeric error code returned by many functions. A zero value indicates success; a nonzero value indicates error.

Definition

```
typedef enum {
   XMLERR_OK              = 0, /* success return */
   XMLERR_NULL_PTR        = 1, /* NULL pointer */
   XMLERR_NO_MEMORY       = 2, /* out of memory */
   XMLERR_HASH_DUP        = 3, /* duplicate entry in hash table */
   XMLERR_INTERNAL        = 4,  /* internal error */
   XMLERR_BUFFER_OVERFLOW = 5,  /* name/quoted string too long */
   XMLERR_BAD_CHILD       = 6,  /* invalid child for parent */
   XMLERR_EOI             = 7,  /* unexpected EndOfInformation */
   XMLERR_BAD_MEMCB       = 8,  /* invalid memory callbacks */
   XMLERR_UNICODE_ALIGN   = 12, /* Unicode data misalignment */
   XMLERR_NODE_TYPE       = 13, /* wrong node type */
   XMLERR_UNCLEAN         = 14, /* context is not clean */
   XMLERR_NESTED_STRINGS  = 18, /* internal: nested open str */
   XMLERR_PROP_NOT_FOUND  = 19, /* property not found */
   XMLERR_SAVE_OVERFLOW   = 20, /* save output overflowed */
   XMLERR_NOT_IMP         = 21, /* feature not implemented */
   XMLERR_NLS_MISMATCH    = 50, /* specify lxglo/lxd or neither*/
   XMLERR_NLS_INIT        = 51, /* error at NLS initialization */
   XMLERR_LEH_INIT        = 52, /* error at LEH initialization */
   XMLERR_LML_INIT        = 53, /* error at LML initialization */
   XMLERR_LPU_INIT        = 54  /* error at LPU initialization */
} xmlerr;
```

xmlistream

Generic user-defined input stream. The three function pointers are required (but may be stubs). The context pointer is entirely user-defined; point it to whatever state information is required to manage the stream; it will be passed as first argument to the user functions.

Definition

```
typedef struct xmlistream {
   XML_STREAM_OPEN_F(
      (*open_xmlistream),
```

```
      xctx,
      sctx,
      path,
      parts,
      length);
   XML_STREAM_READ_F(
      (*read_xmlistream),
      xctx,
      sctx,
      path,
      dest,
      size,
      nraw, eoi);
   XML_STREAM_CLOSE_F(
      (*close_xmlistream),
      xctx,
      sctx);
   void *ctx_xmlistream;                          /* user's stream context */
} xmlistream;
```

xmliter

Control structure for DOM 2 `NodeIterator` and `TreeWalker`.

Definition

```
struct xmliter {
   xmlnode *root_xmliter;   /* root node of the iteration space */
   xmlnode *cur_xmliter;    /* current position iterator ref node */
   ub4      show_xmliter;   /* node filter mask */
   void    *filt_xmliter;   /* node filter function */
   boolean  attach_xmliter; /* is iterator valid? */
   boolean  expan_xmliter;  /* are external entities expanded? */
   boolean  before_xmliter; /* iter position before ref node? */
};
typedef struct xmliter xmliter;
typedef struct xmliter xmlwalk;
```

xmlnodetype

The numeric type code of a node. `0` means invalid, 1-13 are the standard numberings from DOM 1.0, and higher numbers are for internal use only.

Definition

```
typedef enum {
    XMLDOM_NONE     = 0,  /* bogus node */
    XMLDOM_ELEM     = 1,  /* element */
    XMLDOM_ATTR     = 2,  /* attribute */
    XMLDOM_TEXT     = 3,  /* char data not escaped by CDATA */
    XMLDOM_CDATA    = 4,  /* char data escaped by CDATA */
    XMLDOM_ENTREF   = 5,  /* entity reference */
    XMLDOM_ENTITY   = 6,  /* entity */
    XMLDOM_PI       = 7,  /* <?processing instructions?> */
    XMLDOM_COMMENT  = 8,  /* <!-- Comments --> */
    XMLDOM_DOC      = 9,  /* Document */
    XMLDOM_DTD      = 10, /* DTD */
    XMLDOM_FRAG     = 11, /* Document fragment */
    XMLDOM_NOTATION = 12, /* notation */

    /* Oracle extensions from here on */
    XMLDOM_ELEMDECL = 13, /* DTD element declaration */
    XMLDOM_ATTRDECL = 14, /* DTD attribute declaration */

    /* Content Particles (nodes in element's Content Model) */
    XMLDOM_CPELEM   = 15, /* element */
    XMLDOM_CPCHOICE = 16, /* choice (a|b) */
    XMLDOM_CPSEQ    = 17, /* sequence (a,b) */
    XMLDOM_CPPCDATA = 18, /* #PCDATA */
    XMLDOM_CPSTAR   = 19, /* '*' (zero or more) */
    XMLDOM_CPPLUS   = 20, /* '+' (one or more) */
    XMLDOM_CPOPT    = 21, /* '?' (optional) */
    XMLDOM_CPEND    = 22  /* end marker */
} xmlnodetype;
```

xmlostream

Generic user-defined output stream. The three function pointers are required (but may be stubs). The context pointer is entirely user-defined; point it to whatever state information is required to manage the stream; it will be passed as first argument to the user functions.

Definition

```
typedef struct xmlostream {
   XML_STREAM_OPEN_F(
      (*open_xmlostream),
      xctx,
```

```
      sctx,
      path,
      parts,
      length);
   XML_STREAM_WRITE_F(
      (*write_xmlostream),
      xctx,
      sctx,
      path,
      src,
      size);
   XML_STREAM_CLOSE_F(
      (*close_xmlostream),
      xctx,
      sctx);
   void *ctx_xmlostream;        /* user's stream context */
} xmlostream;
```

xmlpoint

XPointer point location.

Definition

```
typedef struct xmlpoint xmlpoint;
```

xmlrange

Control structure for DOM 2 Range.

Definition

```
typedef struct xmlrange {
 xmlnode *startnode_xmlrange;  /* start point container */
 ub4      startofst_xmlrange;  /* start point index */
 xmlnode *endnode_xmlrange;    /* end point container */
 ub4      endofst_xmlrange;    /* end point index */
 xmlnode *doc_xmlrange;        /* document node */
 xmlnode *root_xmlrange;       /* root node of the range */
 boolean  collapsed_xmlrange;  /* is range collapsed? */
 boolean  detached_xmlrange;   /* range invalid, invalidated?*/
} xmlrange;
```

xmlshowbits

Bit flags used to select which nodes types to show.

Definition

```
typedef ub4 xmlshowbits;
#define XMLDOM_SHOW_ALL        ~(ub4)0
#define XMLDOM_SHOW_BIT(ntype) ((ub4)1 << (ntype))
#define XMLDOM_SHOW_ELEM       XMLDOM_SHOW_BIT(XMLDOM_ELEM)
#define XMLDOM_SHOW_ATTR       XMLDOM_SHOW_BIT(XMLDOM_ATTR)
#define XMLDOM_SHOW_TEXT       XMLDOM_SHOW_BIT(XMLDOM_TEXT)
#define XMLDOM_SHOW_CDATA      XMLDOM_SHOW_BIT(XMLDOM_CDATA)
#define XMLDOM_SHOW_ENTREF     XMLDOM_SHOW_BIT(XMLDOM_ENTREF)
#define XMLDOM_SHOW_ENTITY     XMLDOM_SHOW_BIT(XMLDOM_ENTITY)
#define XMLDOM_SHOW_PI         XMLDOM_SHOW_BIT(XMLDOM_PI)
#define XMLDOM_SHOW_COMMENT    XMLDOM_SHOW_BIT(XMLDOM_COMMENT)
#define XMLDOM_SHOW_DOC        XMLDOM_SHOW_BIT(XMLDOM_DOC)
#define XMLDOM_SHOW_DTD        XMLDOM_SHOW_BIT(XMLDOM_DTD)
#define XMLDOM_SHOW_FRAG       XMLDOM_SHOW_BIT(XMLDOM_FRAG)
#define XMLDOM_SHOW_NOTATION   XMLDOM_SHOW_BIT(XMLDOM_NOTATION)
#define XMLDOM_SHOW_DOC_TYPE   XMLDOM_SHOW_BIT(XMLDOM_DOC_TYPE)
```

xmlurlacc

This is an enumeration of the known access methods for retrieving data from a URL. Open/read/close functions may be plugged in to override the default behavior.

Definition

```
typedef enum {
    XML_ACCESS_NONE    = 0, /* not specified */
    XML_ACCESS_UNKNOWN = 1, /* specified but unknown */
    XML_ACCESS_FILE    = 2, /* filesystem access */
    XML_ACCESS_HTTP    = 3, /* HTTP */
    XML_ACCESS_FTP     = 4, /* FTP */
    XML_ACCESS_GOPHER  = 5, /* Gopher */
    XML_ACCESS_ORADB   = 6, /* Oracle DB */
    XML_ACCESS_STREAM  = 7  /* user-defined stream */
} xmlurlacc;
```

xmlurlhdl

This union contains the handle(s) needed to access URL data, be it a stream or stdio pointer, file descriptor(s), and so on.

Definition

```
typedef union xmlurlhdl {
    void  *ptr_xmlurlhdl;  /* generic stream/file/... handle */
    struct {
        sb4 fd1_xmlurlhdl; /* file descriptor(s) [FTP needs all 3!] */
        sb4 fd2_xmlurlhdl;
        sb4 fd3_xmlurlhdl;
    } fds_lpihdl;
} xmlurlhdl;
```

xmlurlpart

This structure contains the sub-parts of a URL. The original URL is parsed and the pieces copies (NULL-terminated) to a working buffer, then this structure is filled in to point to the parts. Given URL `http://user:pwd@baz.com:8080/pub/baz.html;quux=1?huh#fraggy`, the example component part from this URL will be shown.

Definition

```
typedef struct xmlurlpart {
 xmlurlacc access_xmlurlpart;   /* access method code, XMLACCESS_HTTP */
 oratext   *accbuf_xmlurlpart;  /* access method name: "http" */
 oratext   *host_xmlurlpart;    /* hostname:          "baz.com" */
 oratext   *dir_xmlurlpart;     /* directory:         "pub" */
 oratext   *file_xmlurlpart;    /* filename:          "baz.html" */
 oratext   *uid_xmlurlpart;     /* userid/username:   "user" */
 oratext   *passwd_xmlurlpart;  /* password:          "pwd" */
 oratext   *port_xmlurlpart;    /* port (as string): "8080" */
 oratext   *frag_xmlurlpart;    /* fragment:          "fraggy" */
 oratext   *query_xmlurlpart;   /* query:             "huh" */
 oratext   *param_xmlurlpart;   /* parameter:         "quux=1" */
 ub2        portnum_xmlurlpart; /* port (as number): 8080 */
 ub1        abs_xmlurlpart;     /* absolute path?     TRUE */
} xmlurlpart;
```

xmlxptrloc

XPointer location data type.

Definition

```
typedef struct xmlxptrloc xmlxptrloc;
```

xmlxptrlocset

XPointer location set data type.

Definition

```
typedef struct xmlxptrlocset xmlxptrlocset;
```

xmlxslobjtype

Type of XSLT object that may be returned.

Definition

```
typedef enum xmlxslobjtype {
    XMLXSL_TYPE_UNKNOWN = 0, /* Not a defined type */
    XMLXSL_TYPE_NDSET   = 1, /* Node-set */
    XMLXSL_TYPE_BOOL    = 2, /* Boolean value */
    XMLXSL_TYPE_NUM     = 3, /* Numeric value (double) */
    XMLXSL_TYPE_STR     = 4, /* String */
    XMLXSL_TYPE_FRAG    = 5  /* Document Fragment */
} xmlxslobjtype;
```

xmlxslomethod

Type of output to be produced by the XSLT processor.

Definition

```
typedef enum xmlxslomethod {
 XMLXSL_OUTPUT_UNKNOWN = 0, /* Not defined */
 XMLXSL_OUTPUT_XML     = 1, /* Produce a Document Fragment */
 XMLXSL_OUTPUT_STREAM  = 2, /* Stream out formatted result */
 XMLXSL_OUTPUT_HTML    = 3  /* Stream out HTML formatted result */
```

```
} xmlxslomethod;
```

xmlxvm

An object of type `xmlxvm` is used for XML documents transformation. The contents of `xmlxvm` are private and must not be accessed by users.

Definition

```
struct xmlxvm;
typedef struct xmlxvm xmlxvm;
```

xmlxvmcomp

An object of type `xmlxvmcomp` is used for compiling XSL stylesheets. The contents of `xmlxvmcomp` are private and must not be accessed by users.

Definition

```
struct xmlxvmcomp;
typedef struct xmlxvmcomp xmlxvmcomp;
```

xmlxvmflags

Control flags for the XSLT compiler.

- `XMLXVM_DEBUG` forces compiler to insert debug information into the bytecode.
- `XMLXVM_STRIPSPACE` forces the same behavior as `xsl:strip-space elements="*"`

Definition

```
typedef ub4 xmlxvmflag;
#define XMLXVM_NOFLAG     0x00
#define XMLXVM_DEBUG      0x01 /* insert debug info into bytecode */
#define XMLXVM_STRIPSPACE 0x02 /* same as xsl:strip-space elements="*" */
```

xmlxvmobjtype

Type of XSLTVM object.

Definition

```
typedef enum xmlxvmobjtype {
    XMLXVM_TYPE_UNKNOWN = 0,
    XMLXVM_TYPE_NDSET   = 1,
    XMLXVM_TYPE_BOOL    = 2,
    XMLXVM_TYPE_NUM     = 3,
    XMLXVM_TYPE_STR     = 4,
    XMLXVM_TYPE_FRAG    = 5
} xmlxvmobjtype;
```

xpctx

XPath top-level context.

Definition

```
struct xpctx;
typedef struct xpctx xpctx;
```

xpexpr

XPath expression.

Definition

```
struct xpexpr;
typedef struct xpexpr xpexpr;
```

xpobj

Xpath object.

Definition

```
struct xpobj;
typedef struct xpobj xpobj;
```

xsdctx

XML Schema validator context, created by `XmlSchemaCreate` and passed to most Schema functions.

Definition

```
# define XSDCTX_DEFINED
struct xsdctx; typedef struct xsdctx xsdctx;
```

xslctx

XSL top-level context.

Definition

```
struct xslctx;
typedef struct xslctx xslctx;
```

xvmobj

XSLVM processor run-time object; content is private and must not be accessed by users.

Definition

```
struct xvmobj;
typedef struct xvmobj xvmobj;
```

3

Package Callback APIs for C

This package defines macros which declare functions (or function pointers) for XML callbacks. Callbacks are used for error-message handling, memory allocation and freeing, and stream operations.

This chapter contains the following section:

- Callback Methods

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Callback Methods

Table 3–1 summarizes the methods of available through the Callback interface.

Table 3–1 Summary of Callback Methods

Function	Summary
XML_ACCESS_CLOSE_F on page 3-2	User-defined access method close callback.
XML_ACCESS_OPEN_F on page 3-3	User-defined access method open callback.
XML_ACCESS_READ_F on page 3-3	User-defined access method read callback.
XML_ALLOC_F on page 3-4	Low-level memory allocation.
XML_ERRMSG_F on page 3-5	Handle error message.
XML_FREE_F on page 3-5	Low-level memory freeing.
XML_STREAM_CLOSE_F on page 3-6	User-defined stream close callback.
XML_STREAM_OPEN_F on page 3-6	User-defined stream open callback.
XML_STREAM_READ_F on page 3-7	User-defined stream read callback.
XML_STREAM_WRITE_F on page 3-8	User-defined stream write callback.

XML_ACCESS_CLOSE_F

This macro defines a prototype for the close function callback used to access a URL.

Syntax

```
#define XML_ACCESS_CLOSE_F(func, ctx, uh)
xmlerr func(
   void *ctx,
    xmlurlhdl *uh)
```

Parameter	In/Out	Description
ctx	IN	user-defined context
uh	IN	URL handle(s)

Returns

`(xmlerr)` numeric error code, `0` on success

See Also: XML_ACCESS_OPEN_F, XML_ACCESS_READ_F

XML_ACCESS_OPEN_F

This macro defines a prototype for the open function callback used to access a URL.

Syntax

```
#define XML_ACCESS_OPEN_F(func, ctx, uri, parts, length, uh)
xmlerr func(
   void *ctx,
   oratext *uri,
   xmlurlpart *parts,
   ubig_ora *length,
   xmlurlhdl *uh)
```

Parameter	In/Out	Description
`ctx`	`IN`	user-defined context
`uri`	`IN`	URI to be opened
`parts`	`IN`	URI broken into components
`length`	`OUT`	total length of input data if known, `0` otherwise
`uh`	`IN`	URL handle(s)

Returns

`(xmlerr)` numeric error code, `0` on success

See Also: XML_ACCESS_CLOSE_F, XML_ACCESS_READ_F

XML_ACCESS_READ_F

This macro defines a prototype for the read function callback used to access a URL.

Syntax

```
#define XML_ACCESS_READ_F(func, ctx, uh, data, nraw, eoi)
xmlerr func(
   void *ctx,
   xmlurlhdl *uh,
   oratext **data,
   ubig_ora *nraw,
   ub1 *eoi)
```

Parameter	In/Out	Description
ctx	IN	user-defined context
uh	IN	URL handle(s)
data	IN/OUT	recipient data buffer; reset to start of data
nraw	OUT	number of real data bytes read
eoi	OUT	signal to end of information; last chunk

Returns

(xmlerr) numeric error code, 0 on success

See Also: XML_ACCESS_OPEN_F, XML_ACCESS_CLOSE_F

XML_ALLOC_F

This macro defines a prototype for the low-level memory alloc function provided by the user. If no allocator is provided, malloc is used. Memory should not be zeroed by this function. Matches XML_FREE_F.

Syntax

```
#define XML_ALLOC_F(func, mctx, size)
void *func(
   void *mctx,
   size_t size)
```

Parameter	In/Out	Description
mctx	IN	low-level memory context

Parameter	In/Out	Description
size	IN	number of bytes to allocated

Returns

(void *) allocated memory

See Also: XML_FREE_F

XML_ERRMSG_F

This macro defines a prototype for the error message handling function. If no error message callback is provided at XML initialization time, errors will be printed to stderr. If a handler is provided, it will be invoked instead of printing to stderr.

Syntax

```
#define XML_ERRMSG_F(func, ectx, msg, err)
void func(
   void *ectx,
   oratext *msg,
   xmlerr err)
```

Parameter	In/Out	Description
ectx	IN	error message context
msg	IN	text of error message
err	IN	numeric error code

See Also: XmlCreate in Chapter 9, "Package XML APIs for C"

XML_FREE_F

This macro defines a prototype for the low-level memory free function provided by the user. If no allocator is provided, free() is used. Matches XML_ALLOC_F.

Syntax

```
#define XML_FREE_F(func, mctx, ptr)
void func(
   void *mctx,
   void *ptr)
```

Parameter	In/Out	Description
mctx	IN	low-level memory context
ptr	IN	memory to be freed

XML_STREAM_CLOSE_F

This macro defines a prototype for the close function callback, called to close an open source and free its resources.

Syntax

```
#define XML_STREAM_CLOSE_F(func, xctx, sctx)
void func(
   xmlctx *xctx,
   void *sctx)
```

Parameter	In/Out	Description
xctx	IN	XML context
sctx	IN	user-defined stream context

See Also: XML_STREAM_OPEN_F, XML_STREAM_READ_F, XML_STREAM_WRITE_F

XML_STREAM_OPEN_F

This macro defines a prototype for the open function callback, which is called once to open the input source. This function should return XMLERR_OK on success.

Syntax

```
#define XML_STREAM_OPEN_F(func, xctx, sctx, path, parts, length)
xmlerr func(
   xmlctx *xctx,
   void *sctx,
   oratext *path,
   void *parts,
   ubig_ora *length)
```

Parameter	In/Out	Description
xctx	IN	XML context
sctx	IN	user-defined stream context
path	IN	full path of the URI to be opened
parts	IN	URI broken down into components (opaque pointer)
length	(OUT)	total length of input data if known, 0 if not known

Returns

(xmlerr) numeric error code, 0 on success

See Also: XML_STREAM_CLOSE_F, XML_STREAM_READ_F, XML_STREAM_WRITE_F

XML_STREAM_READ_F

This macro defines a prototype for the read function callback, called to read data from an open source into a buffer, returning the number of bytes read (< 0 on error). The eoi flag determines if this is the final block of data.

On EOI, the close function will be called automatically.

Syntax

```
#define XML_STREAM_READ_F(func, xctx, sctx, path, dest, size, nraw, eoi)
xmlerr func(
   xmlctx *xctx,
   void *sctx,
   oratext *path,
   oratext *dest,
```

```
   size_t size,
   sbig_ora *nraw,
   boolean *eoi)
```

Parameter	In/Out	Description
xctx	IN	XML context
sctx	IN	user-defined stream context
path	IN	full URI of the open source (for error messages)
dest	(OUT)	destination buffer to read data into
size	IN	size of destination buffer
nraw	(OUT)	number of bytes read
eoi	(OUT)	signal to end of information; last chunk

Returns

(xmlerr) numeric error code, 0 on success

See Also: XML_STREAM_OPEN_F, XML_STREAM_CLOSE_F, XML_STREAM_WRITE_F

XML_STREAM_WRITE_F

This macro defines a prototype for the write function callback, called to write data to a user-defined stream.

Syntax

```
#define XML_STREAM_WRITE_F(func, xctx, sctx, path, src, size)
xmlerr func(
   xmlctx *xctx,
   void *sctx,
   oratext *path,
   oratext *src,
   size_t size)
```

Parameter	In/Out	Description
xctx	IN	XML context
sctx	IN	user-defined stream context
path	IN	full URI of the open source (for error messages)
src	IN	source buffer to read data from
size	IN	size of source in bytes

Returns

(xmlerr) numeric error code, 0 on success

See Also: XML_STREAM_OPEN_F, XML_STREAM_CLOSE_F, XML_STREAM_READ_F

4

Package DOM APIs for C

This implementation follows REC-DOM-Level-1-19981001. Because the DOM standard is object-oriented, some changes were made for C language adaptation.

- Reused function names have to be expanded; `getValue` in the `Attr` interface has the unique name `XmlDomGetAttrValue` that matches the pattern established by DOM 2's `getNodeValue`.
- Functions were added to extend the DOM beyond the standard; one example is `XmlDomNumChildNodes`, which returns the number of children of a node.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Attr Interface

Table 4–1 summarizes the methods of available through the `Attr` interface.

Table 4–1 Summary of Attr Methods; DOM Package

Function	Summary
XmlDomGetAttrLocal on page 4-3	Returns an attribute's namespace local name as `NULL`-terminated string.
XmlDomGetAttrLocalLen on page 4-3	Returns an attribute's namespace local name as length-encoded string.
XmlDomGetAttrName on page 4-4	Return attribute's name as `NULL`-terminated string.
XmlDomGetAttrNameLen on page 4-5	Return attribute's name as length-encoded string.
XmlDomGetAttrPrefix on page 4-6	Returns an attribute's namespace prefix.
XmlDomGetAttrSpecified on page 4-7	Return flag that indicates whether an attribute was explicitly created.
XmlDomGetAttrURI on page 4-7	Returns an attribute's namespace URI as `NULL`-terminated string.
XmlDomGetAttrURILen on page 4-8	Returns an attribute's namespace URI as length-encoded string.
XmlDomGetAttrValue on page 4-9	Return attribute's value as `NULL`-terminated string.
XmlDomGetAttrValueLen on page 4-10	Return attribute's value as length-encoded string.
XmlDomGetAttrValueStream on page 4-11	Get attribute value stream-style,i.e.chunked.
XmlDomGetOwnerElem on page 4-11	Return an attribute's "owning" element.
XmlDomSetAttrValue on page 4-12	Set an attribute's value.
XmlDomSetAttrValueStream on page 4-13	Sets an attribute value stream style (chunked).

XmlDomGetAttrLocal

Returns an attribute's namespace local name (in the data encoding). If the attribute's name is not fully qualified (has no prefix), then the local name is the same as the name.

A length-encoded version is available as `XmlDomGetAttrURILen` which returns the local name as a pointer and length, for use if the data is known to use `XMLType` backing store.

Syntax

```
oratext* XmlDomGetAttrLocal(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`attr`	`IN`	attribute node

Returns

`(oratext *)` attribute's local name [data encoding]

See Also: XmlDomGetAttrLocalLen, XmlDomGetAttrName, XmlDomGetAttrURI, XmlDomGetAttrPrefix

XmlDomGetAttrLocalLen

Returns an attribute's namespace local name (in the data encoding). If the attribute's name is not fully qualified (has no prefix), then the local name is the same as the name.

A `NULL`-terminated version is available as `XmlDomGetAttrLocal` which returns the local name as `NULL`-terminated string. If the backing store is known to be `XMLType`, then the attribute's data will be stored internally as length-encoded. Using the length-based `Get`*`XXX`* functions will avoid having to copy and `NULL`-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetAttrLocalLen(
   xmlctx *xctx,
   xmlattrnode *attr,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of local name, in characters

Returns

(oratext *) Attr's local name [data encoding]

See Also: XmlDomGetAttrLocal, XmlDomGetAttrName, XmlDomGetAttrURI, XmlDomGetAttrPrefix

XmlDomGetAttrName

Returns the fully-qualified name of an attribute (in the data encoding) as a NULL-terminated string, for example bar\0 or foo:bar\0.

A length-encoded version is available as XmlDomGetAttrNameLen which returns the attribute name as a pointer and length, for use if the data is known to use XMLType backing store.

Syntax

```
oratext* XmlDomGetAttrName(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node

Returns

(oratext *) name of attribute [data encoding]

See Also: XmlDomGetAttrNameLen, XmlDomGetAttrURI, XmlDomGetAttrPrefix, XmlDomGetAttrLocal

XmlDomGetAttrNameLen

Returns the fully-qualified name of an attribute (in the data encoding) as a length-encoded string, for example ("bar", 3) or ("foo:bar", 7).

A NULL-terminated version is available as XmlDomGetAttrName which returns the attribute name as NULL-terminated string. If the backing store is known to be XMLType, then the attribute's data will be stored internally as length-encoded. Using the length-based Get*XXX* functions will avoid having to copy and NULL-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetAttrNameLen(
   xmlctx *xctx,
   xmlattrnode *attr,
   oratext *buf,
   ub4 buflen,
```

```
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of local name, in characters

Returns

`(oratext *)` name of attribute [data encoding]

> **See Also:** XmlDomGetAttrName, XmlDomGetAttrURI, XmlDomGetAttrPrefix, XmlDomGetAttrLocal

XmlDomGetAttrPrefix

Returns an attribute's namespace prefix (in the data encoding). If the attribute's name is not fully qualified (has no prefix), `NULL` is returned.

Syntax

```
oratext* XmlDomGetAttrPrefix(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node

Returns

`(oratext *)` attribute's namespace prefix [data encoding] or `NULL`

See Also: XmlDomGetAttrName, XmlDomGetAttrURI, XmlDomGetAttrLocal

XmlDomGetAttrSpecified

Return the 'specified' flag for an attribute. If the attribute was explicitly given a value in the original document, this is `TRUE`; otherwise, it is `FALSE`. If the node is not an attribute, returns `FALSE`. If the user sets an attribute's value through DOM, its specified flag will be `TRUE`. To return an attribute to its default value (if it has one), the attribute should be deleted; it will then be re-created automatically with the default value (and specified will be `FALSE`).

Syntax

```
boolean XmlDomGetAttrSpecified(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`attr`	`IN`	attribute node

Returns

`(boolean)` attribute's "specified" flag

See Also: XmlDomSetAttrValue

XmlDomGetAttrURI

Returns an attribute's namespace URI (in the data encoding). If the attribute's name is not qualified (does not contain a namespace prefix), it will have the default namespace in effect when the node was created (which may be `NULL`).

A length-encoded version is available as `XmlDomGetAttrURILen` which returns the URI as a pointer and length, for use if the data is known to use `XMLType` backing store.

Syntax

```
oratext* XmlDomGetAttrURI(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node

Returns

(oratext *) attribute's namespace URI [data encoding] or NULL

See Also: XmlDomGetAttrURILen, XmlDomGetAttrPrefix, XmlDomGetAttrLocal

XmlDomGetAttrURILen

Returns an attribute's namespace URI (in the data encoding) as length-encoded string. If the attribute's name is not qualified (does not contain a namespace prefix), it will have the default namespace in effect when the node was created (which may be NULL).

A NULL-terminated version is available as XmlDomGetAttrURI which returns the URI as NULL-terminated string. If the backing store is known to be XMLType, then the attribute's data will be stored internally as length-encoded. Using the length-based Get functions will avoid having to copy and NULL-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetAttrURILen(
   xmlctx *xctx,
   xmlattrnode *attr,
   oratext *buf,
```

```
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of URI, in characters

Returns

(oratext *) attribute's namespace URI [data encoding] or NULL

See Also: XmlDomGetAttrURI, XmlDomGetAttrPrefix, XmlDomGetAttrLocal

XmlDomGetAttrValue

Returns the "value" (character data) of an attribute (in the data encoding) as NULL-terminated string. Character and general entities will have been replaced.

A length-encoded version is available as XmlDomGetAttrValueLen which returns the attribute value as a pointer and length, for use if the data is known to use XMLType backing store.

Syntax

```
oratext* XmlDomGetAttrValue(
   xmlctx *xctx,
   xmlattrnode *attr)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node

Returns

`(oratext *)` attribute's value

See Also: XmlDomGetAttrValueLen, XmlDomSetAttrValue

XmlDomGetAttrValueLen

Returns the "value" (character data) of an attribute (in the data encoding) as length-encoded string. Character and general entities will have been replaced.

A `NULL`-terminated version is available as `XmlDomGetAttrValue` which returns the attribute value as `NULL`-terminated string. If the backing store is known to be `XMLType`, then the attribute's data will be stored internally as length-encoded. Using the length-based `Get`*`XXX`* functions will avoid having to copy and `NULL`-terminate the data.

If both the input buffer is non-`NULL` and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetAttrValueLen(
   xmlctx *xctx,
   xmlattrnode *attr,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`attr`	`IN`	attribute node
`buf`	`IN`	input buffer; optional
`buflen`	`IN`	input buffer length; optional
`len`	`OUT`	length of attribute's value, in characters

Returns

`(oratext *)` attribute's value

See Also: XmlDomGetAttrValue, XmlDomSetAttrValue

XmlDomGetAttrValueStream

Returns the large "value" (associated character data) for an attribute and sends it in pieces to the user's output stream. For very large values, it is not always possible to store them [efficiently] as a single contiguous chunk. This function is used to access chunked data of that type.

Syntax

```
xmlerr XmlDomGetAttrValueStream(
   xmlctx *xctx,
   xmlnode *attr,
   xmlostream *ostream)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`attr`	`IN`	attribute node
`ostream`	`IN`	output stream object

Returns

`(xmlerr)` numeric error code, `0` on success

XmlDomGetOwnerElem

Returns the `Element` node associated with an attribute. Each attr either belongs to an element (one and only one), or is detached and not yet part of the DOM tree. In the former case, the element node is returned; if the attr is unassigned, `NULL` is returned.

Syntax

```
xmlelemnode* XmlDomGetOwnerElem(
   xmlctx *xctx,
```

```
   xmlattrnode *attr)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node

Returns

`(xmlelemnode *)` attribute's element node [or `NULL`]

See Also: XmlDomGetOwnerDocument

XmlDomSetAttrValue

Sets the given attribute's value to data. If the node is not an attribute, does nothing. Note that the new value must be in the data encoding! It is not verified, converted, or checked. The attribute's specified flag will be `TRUE` after setting a new value.

Syntax

```
void XmlDomSetAttrValue(
   xmlctx *xctx,
   xmlattrnode *attr,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node
value	IN	new value of attribute; data encoding

See Also: XmlDomGetAttrValue

XmlDomSetAttrValueStream

Sets the large "value" (associated character data) for an attribute piecemeal from an input stream. For very large values, it is not always possible to store them efficiently as a single contiguous chunk. This function is used to access chunked data of that type.

Syntax

```
xmlerr XmlDomSetAttrValueStream(
   xmlctx *xctx,
   xmlnode *attr,
   xmlistream *istream)
```

Parameter	In/Out	Description
xctx	IN	XML context
attr	IN	attribute node
isream	IN	input stream

Returns

(xmlerr) numeric error code, 0 on success

CharacterData Interface

Table 4–2 summarizes the methods of available through the `CharacterData` interface.

Table 4–2 Summary of CharacterData Method; DOM Package

Function	Summary
XmlDomAppendData on page 4-14	Append data to end of node's current data.
XmlDomDeleteData on page 4-15	Remove part of node's data.
XmlDomGetCharData on page 4-16	Return data for node.
XmlDomGetCharDataLength on page 4-16	Return length of data for node.
XmlDomInsertData on page 4-17	Insert string into node's current data.
XmlDomReplaceData on page 4-18	Replace part of node's data.
XmlDomSetCharData on page 4-19	Set data for node.
XmlDomSubstringData on page 4-19	Return substring of node's data.

XmlDomAppendData

Append a string to the end of a `CharacterData` node's data. If the node is not `Text`, `Comment` or `CDATA`, or if the string to append is `NULL`, does nothing. The appended data should be in the data encoding. It will not be verified, converted, or checked.

The new node data will be allocated and managed by DOM, but if the previous node value was allocated and manager by the user, they are responsible for freeing it, which is why it is returned.

Syntax

```
void XmlDomAppendData(
   xmlctx *xctx,
   xmlnode *node,
   oratext *data,
   oratext **old)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node
data	IN	data to append; data encoding
old	OUT	previous data for node; data encoding

See Also: XmlDomGetCharData, XmlDomInsertData, XmlDomDeleteData, XmlDomReplaceData, XmlDomSplitText

XmlDomDeleteData

Remove a range of characters from a CharacterData node's data. If the node is not text, comment or CDATA, or if the offset is outside of the original data, does nothing. The offset is zero-based, so offset zero refers to the start of the data. Both offset and count are in characters, not bytes. If the sum of offset and count exceeds the data length then all characters from offset to the end of the data are deleted.

The new node data will be allocated and managed by DOM, but if the previous node value was allocated and managed by the user, they are responsible for freeing it, which is why it is returned.

Syntax

```
void XmlDomDeleteData(
   xmlctx *xctx,
   xmlnode *node,
   ub4 offset,
   ub4 count,
   oratext **old)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node
offset	IN	character offset where to start removing

Parameter	In/Out	Description
count	IN	number of characters to delete
old	OUT	previous data for node; data encoding

See Also: XmlDomGetCharData, XmlDomAppendData, XmlDomInsertData, XmlDomReplaceData, XmlDomSplitText

XmlDomGetCharData

Returns the data for a CharacterData node (type text, comment or CDATA) in the data encoding. For other node types, or if there is no data, returns NULL.

Syntax

```
oratext* XmlDomGetCharData(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment or CDATA

Returns

(oratext *) character data of node [data encoding]

See Also: XmlDomSetCharData, XmlDomCreateText, XmlDomCreateComment, XmlDomCreateCDATA

XmlDomGetCharDataLength

Returns the length of the data for a CharacterData node, type Text, Comment or CDATA) in characters, not bytes. For other node types, returns 0.

Syntax

```
ub4 XmlDomGetCharDataLength(
```

```
   xmlctx *xctx,
   xmlnode *cdata)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment or CDATA

Returns

(ub4) length in characters, not bytes, of node's data

See Also: XmlDomGetCharData

XmlDomInsertData

Insert a string into a CharacterData node's data at the specified position. If the node is not Text, Comment or CDATA, or if the data to be inserted is NULL, or the offset is outside the original data, does nothing. The inserted data must be in the data encoding. It will not be verified, converted, or checked. The offset is specified as characters, not bytes. The offset is zero-based, so inserting at offset zero prepends the data.

The new node data will be allocated and managed by DOM, but if the previous node value was allocated and managed by the user, they are responsible for freeing it (which is why it's returned).

Syntax

```
void XmlDomInsertData(
   xmlctx *xctx,
   xmlnode *node,
   ub4 offset,
   oratext *arg,
   oratext **old)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment, or CDATA

Parameter	In/Out	Description
offset	IN	character offset where to start inserting
arg	IN	data to insert
old	OUT	previous data for node; data encoding

See Also: XmlDomGetCharData, XmlDomAppendData, XmlDomDeleteData, XmlDomReplaceData, XmlDomSplitText

XmlDomReplaceData

Replaces a range of characters in a CharacterData node's data with a new string. If the node is not text, comment or CDATA, or if the offset is outside of the original data, or if the replacement string is NULL, does nothing. If the count is zero, acts just as XmlDomInsertData. The offset is zero-based, so offset zero refers to the start of the data. The replacement data must be in the data encoding. It will not be verified, converted, or checked. The offset and count are both in characters, not bytes. If the sum of offset and count exceeds length, then all characters to the end of the data are replaced.

The new node data will be allocated and managed by DOM, but if the previous node value was allocated and managed by the user, they are responsible for freeing it, which is why it is returned.

Syntax

```
void XmlDomReplaceData(
   xmlctx *xctx,
   xmlnode *node,
   ub4 offset,
   ub4 count,
   oratext *arg,
   oratext **old)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment, or CDATA
offset	IN	character offset where to start replacing

Parameter	In/Out	Description
count	IN	number of characters to replace
arg	IN	replacement substring; data encoding
old	OUT	previous data for node; data encoding

See Also: XmlDomGetCharData, XmlDomAppendData, XmlDomInsertData, XmlDomDeleteData, XmlDomSplitText

XmlDomSetCharData

Sets data for a CharacterData node (type text, comment or CDATA), replacing the old data. For other node types, does nothing. The new data is not verified, converted, or checked; it should be in the data encoding.

Syntax

```
void XmlDomSetCharData(
   xmlctx *xctx,
   xmlnode *node,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment, or CDATA
data	IN	new data for node

See Also: XmlDomGetCharData

XmlDomSubstringData

Returns a range of character data from a CharacterData node, type Text, Comment or CDATA. For other node types, or if count is zero, returns NULL. Since the data is in the data encoding, offset and count are in characters, not bytes. The

beginning of the string is offset 0. If the sum of offset and count exceeds the length, then all characters to the end of the data are returned.

The substring is permanently allocated in the node's document's memory pool. To free the substring, use XmlDomFreeString.

Syntax

```
oratext* XmlDomSubstringData(
   xmlctx *xctx,
   xmlnode *node,
   ub4 offset,
   ub4 count)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	CharacterData node; Text, Comment, or CDATA
offset	IN	character offset where to start extraction of substring
count	IN	number of characters to extract

Returns

(oratext *) specified substring.

See Also: XmlDomAppendData, XmlDomInsertData, XmlDomDeleteData, XmlDomReplaceData, XmlDomSplitText, XmlDomFreeString

Document Interface

Table 4–3 summarizes the methods of available through the `Document` interface.

Table 4–3 Summary of Document Methods; DOM Package

Function	Summary
XmlDomCreateAttr on page 4-22	Create attribute node.
XmlDomCreateAttrNS on page 4-23	Create attribute node with namespace information.
XmlDomCreateCDATA on page 4-24	Create `CDATA` node.
XmlDomCreateComment on page 4-25	Create comment node.
XmlDomCreateElem on page 4-26	Create an element node.
XmlDomCreateElemNS on page 4-27	Create an element node with namespace information.
XmlDomCreateEntityRef on page 4-28	Create entity reference node.
XmlDomCreateFragment on page 4-28	Create a document fragment.
XmlDomCreatePI on page 4-29	Create PI node.
XmlDomCreateText on page 4-30	Create text node.
XmlDomFreeString on page 4-31	Frees a string allocate by `XmlDomSubstringData`, and others.
XmlDomGetBaseURI on page 4-32	Returns the base URI for a document.
XmlDomGetDTD on page 4-32	Get DTD for document.
XmlDomGetDecl on page 4-33	Returns a document's `XMLDecl` information.
XmlDomGetDocElem on page 4-34	Get top-level element for document.
XmlDomGetDocElemByID on page 4-34	Get document element given ID.
XmlDomGetDocElemsByTag on page 4-35	Obtain document elements.
XmlDomGetDocElemsByTagNS on page 4-36	Obtain document elements (namespace aware version).
XmlDomGetLastError on page 4-37	Return last error code for document.
XmlDomGetSchema on page 4-37	Returns URI of schema associated with document.

Table 4–3 (Cont.) Summary of Document Methods; DOM Package

Function	Summary
XmlDomImportNode on page 4-38	Import a node from another DOM.
XmlDomIsSchemaBased on page 4-39	Indicate whether a schema is associated with a document.
XmlDomSaveString on page 4-40	Saves a string permanently in a document's memory pool.
XmlDomSaveString2 on page 4-40	Saves a Unicode string permanently in a document's memory pool.
XmlDomSetDTD on page 4-42	Sets DTD for document.
XmlDomSetDocOrder on page 4-43	Set document order for all nodes.
XmlDomSetLastError on page 4-43	Sets last error code for document.
XmlDomSync on page 4-44	Synchronizes the persistent version of a document with its DOM.

XmlDomCreateAttr

Creates an attribute node with the given name and value (in the data encoding). Note this function differs from the DOM specification, which does not allow the initial value of the attribute to be set (see `XmlDomSetAttrValue`). The name is required, but the value may be `NULL`; neither is verified, converted, or checked.

This is the non-namespace aware function (see `XmlDomCreateAttrNS`): the new attribute will have `NULL` namespace URI and prefix, and its local name will be the same as its name, even if the name specified is a qualified name.

If given an initial value, the attribute's specified flag will be `TRUE`.

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild`, and so on.

See `XmlDomSetAttr` which creates and adds an attribute in a single operation.

The name and value are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlattrnode* XmlDomCreateAttr(
   xmlctx *xctx,
```

```
   xmldocnode *doc,
   oratext *name,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
name	IN	new node's name; data encoding; user control
value	IN	new node's value; data encoding; user control

Returns

`(xmlattrnode *)` new `Attr` node.

See Also: XmlDomSetAttrValue, XmlDomCreateAttrNS, XmlDomSetAttr, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateAttrNS

Creates an attribute node with the given namespace URI and qualified name; this is the namespace-aware version of `XmlDomCreateAttr`. Note this function differs from the DOM specification, which does not allow the initial value of the attribute to be set (see `XmlDomSetAttrValue`). The name is required, but the value may be `NULL`; neither is verified, converted, or checked.

If given an initial value, the attribute's specified flag will be `TRUE`.

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild`, and so on. See `XmlDomSetAttr` which creates and adds an attribute in a single operation.

The URI, qualified name and value are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlattrnode* XmlDomCreateAttrNS(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *uri,
```

```
   oratext *qname,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
uri	IN	node's namespace URI; data encoding; user control
qname	IN	node's qualified name; data encoding; user control
value	IN	new node's value; data encoding; user control

Returns

`(xmlattrnode *)` new `Attr` node.

See Also: XmlDomSetAttrValue, XmlDomCreateAttr, XmlDomSetAttr, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateCDATA

Creates a `CDATASection` node with the given initial data (which should be in the data encoding). A `CDATASection` is considered verbatim and is never parsed; it will not be joined with adjacent `Text` nodes by the normalize operation. The initial data may be `NULL`; if provided, it is not verified, converted, or checked. The name of a `CDATA` node is always "`#cdata-section`".

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild` and so on.

The `CDATA` is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlcdatanode* XmlDomCreateCDATA(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
data	IN	new node's CDATA; data encoding; user control

Returns

(xmlcdatanode *) new CDATA node.

See Also: XmlDomCreateText, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateComment

Creates a Comment node with the given initial data (which must be in the data encoding). The data may be NULL; if provided, it is not verified, converted, or checked. The name of a Comment node is always "#comment".

The new node is an orphan with no parent; it must be added to the DOM tree with XmlDomAppendChild and so on.

The comment data is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlcommentnode* XmlDomCreateComment(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
data	IN	new node's comment; data encoding; user control

Returns

`(xmlcommentnode *)` new `Comment` node.

See Also: XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateElem

Creates an element node with the given tag name (which should be in the data encoding). Note that the tag name of an element is case sensitive. This is the non-namespace aware function: the new node will have `NULL` namespace URI and prefix, and its local name will be the same as its tag name, even if the tag name specified is a qualified name.

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild` and so on.

The `tagname` is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlelemnode* XmlDomCreateElem(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *tagname)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`doc`	`IN`	XML document node
`tagname`	`IN`	new node's name; data encoding; user control

Returns

`(xmlelemnode *)` new `Element` node.

See Also: XmlDomCreateElemNS, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateElemNS

Creates an element with the given namespace URI and qualified name. Note that element names are case sensitive, and the qualified name is required though the URI may be `NULL`. The qualified name will be split into prefix and local parts, retrievable with `XmlDomGetNodePrefix`, `XmlDomGetNodeLocal`, and so on; the tagName will be the full qualified name.

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild` and so on.

The URI and qualified name are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlelemnode* XmlDomCreateElemNS(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *uri,
   oratext *qname)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`doc`	`IN`	XML document node
`uri`	`IN`	new node's namespace URI; data encoding, user control
`qname`	`IN`	new node's qualified name; data encoding; user control

Returns

`(xmlelemnode *)` new `Element` node.

See Also: XmlDomCreateElem, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateEntityRef

Creates an `EntityReference` node; the name (which should be in the data encoding) is the name of the entity to be referenced. The named entity does not have to exist. The name is not verified, converted, or checked.

`EntityReference` nodes are never generated by the parser; instead, entity references are expanded as encountered. On output, an entity reference node will turn into a "&name;" style reference.

The new node is an orphan with no parent; it must be added to the DOM tree with `XmlDomAppendChild`, and so on.

The entity reference name is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlentrefnode* XmlDomCreateEntityRef(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *name)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`doc`	`IN`	XML document node
`name`	`IN`	name of referenced entity; data encoding; user control

Returns

`(xmlentrefnode *)` new `EntityReference` node.

XmlDomCreateFragment

Creates an empty `DocumentFragment` node. A document fragment is treated specially when it is inserted into a DOM tree: the children of the fragment are inserted in order instead of the fragment node itself. After insertion, the fragment node will still exist, but have no children. See `XmlDomInsertBefore`, `XmlDomReplaceChild`, `XmlDomAppendChild`, and so on. The name of a fragment node is always "`#document-fragment`".

Syntax

```
xmlfragnode* XmlDomCreateFragment(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(xmlfragnode *) new empty DocumentFragment node

See Also: XmlDomInsertBefore, XmlDomReplaceChild, XmlDomAppendChild

XmlDomCreatePI

Creates a ProcessingInstruction node with the given target and data (which should be in the data encoding). The data may be NULL initially, and may be changed later (with XmlDomSetPIData), but the target is required and cannot be changed. Note the target and data are not verified, converted, or checked. The name of a PI node is the same as the target.

The new node is an orphan with no parent; it must be added to the DOM tree with XmlDomAppendChild and so on.

The PI's target and data are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlpinode* XmlDomCreatePI(
   xmlctx *xctx
   xmldocnode *doc,
   oratext *target,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
target	IN	new node's target; data encoding; user control
data	IN	new node's data; data encoding; user control

Returns

(xmlpinode *) new PI node.

See Also: XmlDomGetPITarget, XmlDomGetPIData, XmlDomSetPIData, XmlDomCleanNode, XmlDomFreeNode

XmlDomCreateText

Creates a Text node with the given initial data (which must be non-NULL and in the data encoding). The data may be NULL; if provided, it is not verified, converted, checked, or parsed (entities will not be expanded). The name of a fragment node is always "#text". New data for a Text node can be set with XmlDomSetNodeValue; see the CharacterData interface for editing methods.

The new node is an orphan with no parent; it must be added to the DOM tree with XmlDomAppendChild and so on.

The text data is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmltextnode* XmlDomCreateText(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Parameter	In/Out	Description
data	IN	new node's text; data encoding; user control

Returns

(xmltextnode *) new Text node.

See Also: XmlDomCreateCDATA, XmlDomSetNodeValue, XmlDomGetNodeValue, XmlDomSetCharData, XmlDomGetCharData, XmlDomGetCharDataLength, XmlDomSubstringData, XmlDomAppendData, XmlDomInsertData, XmlDomDeleteData, XmlDomReplaceData, XmlDomCleanNode, XmlDomFreeNode

XmlDomFreeString

Frees the string allocated by XmlDomSubstringData or similar functions. Note that strings explicitly saved with XmlDomSaveString are not freeable individually.

Syntax

```
void XmlDomFreeString(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *str)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	document where the string belongs
str	IN	string to free

See Also: XmlDomSaveString, XmlDomSaveString2

XmlDomGetBaseURI

Returns the base URI for a document. Usually only documents that were loaded from a URI will automatically have a base URI; documents loaded from other sources (`stdin`, buffer, and so on) will not naturally have a base URI, but a base URI may have been set for them using `XmlDomSetBaseURI`, for the purposes of resolving relative URIs in inclusion.

Syntax

```
oratext *XmlDomGetBaseURI(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`doc`	`IN`	XML document node

Returns

`(oratext *)` document's base URI [or `NULL`]

See Also: XmlDomSetBaseURI

XmlDomGetDTD

Returns the DTD node associated with current document; if there is no DTD, returns `NULL`. The DTD cannot be edited, but its children may be retrieved with `XmlDomGetChildNodes` as for other node types.

Syntax

```
xmldtdnode* XmlDomGetDTD(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(xmldtdnode *) DTD node for document [or NULL]

> **See Also:** XmlDomSetDTD, XmlCreateDTD in Chapter 9, "Package XML APIs for C", XmlCreateDocument in Chapter 9, "Package XML APIs for C", XmlDomGetDTDName, XmlDomGetDTDEntities, and XmlDomGetDTDNotations

XmlDomGetDecl

Returns the information from a document's XMLDecl. If there is no XMLDecl, returns XMLERR_NO_DECL. Returned are the XML version# ("1.0" or "2.0"), the specified encoding, and the standalone value. If encoding is not specified, NULL will be set. The standalone flag is three-state: < 0 if standalone was not specified, 0 if it was specified and FALSE, > 0 if it was specified and TRUE.

Syntax

```
xmlerr XmlDomGetDecl(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext **ver,
   oratext **enc,
   sb4 *std)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
ver	OUT	XML version
enc	OUT	encoding specification
std	OUT	standalone specification

Returns

(xmlerr) XML error code, perhaps version/encoding/standalone set

XmlDomGetDocElem

Returns the root element (node) of the DOM tree, or NULL if there is none. Each document has only one uppermost Element node, called the root element. It is created after a document is parsed successfully, or manually by XmlDomCreateElem then XmlDomAppendChild, and so on.

Syntax

```
xmlelemnode* XmlDomGetDocElem(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(xmlelemnode *) root element [or NULL]

See Also: XmlDomCreateElem

XmlDomGetDocElemByID

Returns the element node which has the given ID. If no such ID is defined, returns NULL. Note that attributes named "ID" are not automatically of type ID; ID attributes (which can have any name) must be declared as type ID in the DTD.

The given ID should be in the data encoding or it might not match.

Syntax

```
xmlelemnode* XmlDomGetDocElemByID(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *id)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
id	IN	element's unique ID; data encoding

Returns

(xmlelemnode *) matching element.

See Also: XmlDomGetDocElemsByTag, XmlDomGetDocElemsByTagNS

XmlDomGetDocElemsByTag

Returns a list of all elements in the document tree rooted at the root node with a given tag name, in document order (the order in which they would be encountered in a preorder traversal of the tree). If root is NULL, the entire document is searched.

The special name "*" matches all tag names; a NULL name matches nothing. Note that tag names are case sensitive, and should be in the data encoding or a mismatch might occur.

This function is not namespace aware; the full tag names are compared. If two qualified names with two different prefixes both of which map to the same URI are compared, the comparison will fail. See XmlDomGetElemsByTagNS for the namespace-aware version.

The list should be freed with XmlDomFreeNodeList when it is no longer needed.

The list is not live, it is a snapshot. That is, if a new node which matched the tag name were added to the DOM after the list was returned, the list would not automatically be updated to include the node.

Syntax

```
xmlnodelist* XmlDomGetDocElemsByTag(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
name	IN	tagname to match; data encoding; * for all

Returns

(xmlnodelist *) new NodeList containing all matched Elements.

See Also: XmlDomGetDocElemByID, XmlDomGetDocElemsByTagNS, XmlDomFreeNodeList

XmlDomGetDocElemsByTagNS

Returns a list of all elements (in the document tree rooted at the given node) with a given namespace URI and local name, in the order in which they would be encountered in a preorder traversal of the tree. If root is NULL, the entire document is searched.

The URI and local name should be in the data encoding. The special local name "*" matches all local names; a NULL local name matches nothing. Namespace URIs must always match, however, no wildcard is allowed. Note that comparisons are case sensitive. See XmlDomGetDocElemsByTag for the non-namespace aware version.

The list should be freed with XmlDomFreeNodeList when it is no longer needed.

The list is not live, it is a snapshot. That is, if a new node which matched the tag name were added to the DOM after the list was returned, the list would not automatically be updated to include the node.

Syntax

```
xmlnodelist* XmlDomGetDocElemsByTagNS(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
uri	IN	namespace URI to match; data encoding; * matches all
local	IN	local name to match; data encoding; * matches all

Returns

(xmlnodelist *) new NodeList containing all matched Elements.

See Also: XmlDomGetDocElemByID, XmlDomGetDocElemsByTag, XmlDomFreeNodeList

XmlDomGetLastError

Returns the error code of the last error which occurred in the given document.

Syntax

```
xmlerr XmlDomGetLastError(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(xmlerr) numeric error code, 0 if no error

XmlDomGetSchema

Returns URI of schema associated with document, if there is one, else returns NULL. The XmlLoadDom functions take a schema location hint (URI); the schema is used for efficient layout of XMLType data. If a schema was provided at load time, this function returns TRUE.

Syntax

```
oratext* XmlDomGetSchema(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(oratext *) Schema URI or NULL

See Also: XmlDomIsSchemaBased, XmlLoadDom in Chapter 9, "Package XML APIs for C"

XmlDomImportNode

Imports a node from one Document to another. The new node is an orphan and has no parent; it must be added to the DOM tree with XmlDomAppendChild, and so on. The original node is not modified in any way or removed from its document; instead, a new node is created with copies of all the original node's qualified name, prefix, namespace URI, and local name.

As with XmlDomCloneNode, the deep controls whether the children of the node are recursively imported. If FALSE, only the node itself is imported, and it will have no children. If TRUE, all descendents of the node will be imported as well, and an entire new subtree created.

Document and DocumentType nodes cannot be imported. Imported attributes will have their specified flags set to TRUE. Elements will have only their specified attributes imported; non-specified (default) attributes are omitted. New default attributes (for the destination document) are then added.

Syntax

```
xmlnode* XmlDomImportNode(
   xmlctx *xctx,
   xmldocnode *doc,
   xmlctx *nctx,
   xmlnode *node,
```

```
   boolean deep)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
nctx	IN	XML context of imported node
node	IN	node to import
deep	IN	TRUE to import the subtree recursively

Returns

(xmlnode *) newly imported node (in this Document).

See Also: XmlDomCloneNode

XmlDomIsSchemaBased

Returns flag specifying whether there is a schema associated with this document. The XmlLoadDom functions take a schema location hint (URI); the schema is used for efficient layout of XMLType data. If a schema was provided at load time, this function returns TRUE.

Syntax

```
boolean XmlDomIsSchemaBased(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(boolean) TRUE if there is a schema associated with the document

See Also: XmlDomGetSchema, XmlLoadDom in Chapter 9, "Package XML APIs for C"

XmlDomSaveString

Copies the given string into the document's memory pool, so that it persists for the life of the document. The individual string will not be freeable, and the storage will be returned only when the entire document is freed. Works on single-byte or multibyte encodings; for Unicode strings, use `XmlDomSaveString2`.

Syntax

```
oratext* XmlDomSaveString(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *str)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`doc`	`IN`	XML document node
`str`	`IN`	string to save; data encoding; single- or multi-byte only

Returns

`(oratext *)` saved copy of string

See Also: XmlDomSaveString2, XmlFreeDocument in Chapter 9, "Package XML APIs for C"

XmlDomSaveString2

Copies the given string into the document's memory pool, so that it persists for the life of the document. The individual string will not be freeable, and the storage will be returned only when the entire document is free. Works on Unicode strings only; for single-byte or multibyte strings, use `XmlDomSaveString`.

Syntax

```
ub2* XmlDomSaveString2(
   xmlctx *xctx,
   xmldocnode *doc,
   ub2 *ustr)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
ustr	IN	string to save; data encoding; Unicode only

Returns

(ub2 *) saved copy of string

See Also: XmlDomSaveString,XmlFreeDocument in Chapter 9, "Package XML APIs for C"

XmlDomSetBaseURI

Only documents that were loaded from a URI will automatically have a base URI; documents loaded from other sources (stdin, buffer, and so on) will not naturally have a base URI, so this API is used to set a base URI, for the purposes of relative URI resolution in includes. The base URI should be in the data encoding, and a copy will be made.

Syntax

```
xmlerr XmlDomSetBaseURI(
   xmlctx *xctx,
   xmldocnode *doc,
   oratext *uri)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Parameter	In/Out	Description
uri	IN	base URI to set; data encoding

Returns

(xmlerr) XML error code

See Also: XmlDomGetBaseURI

XmlDomSetDTD

Sets the DTD for document. Note this call may only be used for a blank document, before any parsing has taken place. A single DTD can be set for multiple documents, so when a document with a set DTD is freed, the set DTD is not also freed.

Syntax

```
xmlerr XmlDomSetDTD(
   xmlctx *xctx,
   xmldocnode *doc,
   xmldtdnode *dtdnode)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
dtdnode	IN	DocumentType node to set

Returns

(xmlerr) numeric error code, 0 on success

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDEntities, XmlDomGetDTDNotations

XmlDomSetDocOrder

Sets the document order for each node in the current document. Must be called once on the final document before XSLT processing can occur. Note this is called automatically by the XSLT processor, so ordinarily the user need not make this call.

Syntax

```
ub4 XmlDomSetDocOrder(
   xmlctx *xctx,
   xmldocnode *doc,
   ub4 start_id)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node
start_id	IN	string ID number

Returns

(ub4) highest ordinal assigned

XmlDomSetLastError

Sets the Last Error code for the given document. If doc is NULL, sets the error code for the XML context.

Syntax

```
xmlerr XmlDomSetLastError(
   xmlctx *xctx,
   xmldocnode *doc,
   xmlerr errcode)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Parameter	In/Out	Description
errcode	IN	error code to set, 0 to clear error

Returns

(xmlerr) original error code

XmlDomSync

Causes a modified DOM to be written back out to its original source, synchronizing the persistent store and in-memory versions.

Syntax

```
xmlerr XmlDomSync(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	XML document node

Returns

(xmlerr) numeric error code, 0 on success

DocumentType Interface

Table 4–4 summarizes the methods of available through the `DocumentType` interface.

Table 4–4 Summary of DocumentType Methods; DOM Package

Function	Summary
XmlDomGetDTDEntities on page 4-45	Get entities of DTD.
XmlDomGetDTDInternalSubset on page 4-46	Get DTD's internal subset.
XmlDomGetDTDName on page 4-46	Get name of DTD.
XmlDomGetDTDNotations on page 4-47	Get notations of DTD.
XmlDomGetDTDPubID on page 4-48	Get DTD's public ID.
XmlDomGetDTDSysID on page 4-48	Get DTD's system ID.

XmlDomGetDTDEntities

Returns a named node map of general entities defined by the DTD. If the node is not a DTD, or has no general entities, returns `NULL`.

Syntax

```
xmlnamedmap* XmlDomGetDTDEntities(
   xmlctx *xctx,
   xmldtdnode *dtd)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`dtd`	`IN`	DTD node

Returns

`(xmlnamedmap *)` named node map containing entities declared in DTD

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDNotations, XmlDomGetDTDSysID, XmlDomGetDTDInternalSubset

XmlDomGetDTDInternalSubset

Returns the content model for an element. If there is no DTD, returns NULL.

Syntax

```
xmlnode* XmlDomGetDTDInternalSubset(
   xmlctx *xctx,
   xmldtdnode *dtd,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
dtd	IN	DTD node
name	IN	name of Element; data encoding

Returns

(xmlnode *) content model subtree

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDEntities, XmlDomGetDTDNotations, XmlDomGetDTDPubID

XmlDomGetDTDName

Returns a DTD's name (specified immediately after the DOCTYPE keyword), or NULL if the node is not type DTD.

Syntax

```
oratext* XmlDomGetDTDName(
   xmlctx *xctx,
   xmldtdnode *dtd)
```

Parameter	In/Out	Description
xctx	IN	XML context
dtd	IN	DTD node

Returns

(oratext *) name of DTD

See Also: XmlDomGetDTD, XmlDomGetDTDEntities, XmlDomGetDTDNotations, XmlDomGetDTDSysID, XmlDomGetDTDInternalSubset

XmlDomGetDTDNotations

Returns named node map of notations declared by the DTD. If the node is not a DTD or has no Notations, returns NULL.

Syntax

```
xmlnamedmap* XmlDomGetDTDNotations(
   xmlctx *xctx,
   xmldtdnode *dtd)
```

Parameter	In/Out	Description
xctx	IN	XML context
dtd	IN	DTD node

Returns

(xmlnamedmap *) named node map containing notations declared in DTD

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDEntities, XmlDomGetDTDSysID, XmlDomGetDTDInternalSubset

XmlDomGetDTDPubID

Returns a DTD's public identifier.

Syntax

```
oratext* XmlDomGetDTDPubID(
   xmlctx *xctx,
   xmldtdnode *dtd)
```

Parameter	In/Out	Description
xctx	IN	XML context
dtd	IN	DTD node

Returns

`(oratext *)` DTD's public identifier [data encoding]

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDEntities, XmlDomGetDTDSysID, XmlDomGetDTDInternalSubset

XmlDomGetDTDSysID

Returns a DTD's system identifier.

Syntax

```
oratext* XmlDomGetDTDSysID(
   xmlctx *xctx,
   xmldtdnode *dtd)
```

Parameter	In/Out	Description
xctx	IN	XML context
dtd	IN	DTD node

Returns

`(oratext *)` DTD's system identifier [data encoding]

See Also: XmlDomGetDTD, XmlDomGetDTDName, XmlDomGetDTDEntities, XmlDomGetDTDPubID, XmlDomGetDTDInternalSubset

Element Interface

Table 4–5 summarizes the methods of available through the `Element` Interface.

Table 4–5 Summary of Element Methods; DOM Package

Function	Summary
XmlDomGetAttr on page 4-51	Return attribute's value given its name.
XmlDomGetAttrNS on page 4-51	Return attribute's value given its URI and local name.
XmlDomGetAttrNode on page 4-52	Get attribute by name.
XmlDomGetAttrNodeNS on page 4-53	Get attribute by name (namespace aware version).
XmlDomGetChildrenByTag on page 4-53	Get children of element with given tag name (non-namespace aware).
XmlDomGetChildrenByTagNS on page 4-54	Get children of element with tag name (namespace aware version).
XmlDomGetDocElemsByTag on page 4-35	Obtain doc elements.
XmlDomGetDocElemsByTagNS on page 4-36	Obtain doc elements (namespace aware version).
XmlDomGetTag on page 4-56	Return an element node's tag name.
XmlDomHasAttr on page 4-57	Does named attribute exist?
XmlDomHasAttrNS on page 4-58	Does named attribute exist (namespace aware version)?
XmlDomRemoveAttr on page 4-58	Remove attribute with specified name.
XmlDomRemoveAttrNS on page 4-59	Remove attribute with specified URI and local name.
XmlDomRemoveAttrNode on page 4-60	Remove attribute node.
XmlDomSetAttr on page 4-60	Set new attribute for element.
XmlDomSetAttrNS on page 4-61	Set new attribute for element (namespace aware version).
XmlDomSetAttrNode on page 4-62	Set attribute node.
XmlDomSetAttrNodeNS on page 4-62	Set attribute node (namespace aware version).

XmlDomGetAttr

Returns the value of an element's attribute (specified by name). Note that an attribute may have the empty string as its value, but cannot be NULL. If the element does not have an attribute with the given name, NULL is returned.

Syntax

```
oratext* XmlDomGetAttr(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	attribute's name

Returns

(oratext *) named attribute's value [data encoding; may be NULL]

See Also: XmlDomGetAttrNS, XmlDomGetAttrs, XmlDomGetAttrNode

XmlDomGetAttrNS

Returns the value of an element's attribute (specified by URI and local name). Note that an attribute may have the empty string as its value, but cannot be NULL. If the element does not have an attribute with the given name, NULL is returned.

Syntax

```
oratext* XmlDomGetAttrNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
uri	IN	attribute's namespace URI; data encoding
local	IN	attribute's local name; data encoding

Returns

(oratext *) named attribute's value [data encoding; may be NULL]

See Also: XmlDomGetAttr, XmlDomGetAttrs, XmlDomGetAttrNode

XmlDomGetAttrNode

Returns an element's attribute specified by name. If the node is not an element or the named attribute does not exist, returns NULL.

Syntax

```
xmlattrnode* XmlDomGetAttrNode(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	attribute's name; data encoding

Returns

(xmlattrnode *) attribute with the specified name [or NULL]

See Also: XmlDomGetAttrNodeNS, XmlDomGetAttr

XmlDomGetAttrNodeNS

Returns an element's attribute specified by URI and localname. If the node is not an element or the named attribute does not exist, returns NULL.

Syntax

```
xmlattrnode* XmlDomGetAttrNodeNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
uri	IN	attribute's namespace URI; data encoding
local	IN	attribute's local name; data encoding

Returns

(xmlattrnode *) attribute node with the given URI/local name [or NULL]

See Also: XmlDomGetAttrNode, XmlDomGetAttr

XmlDomGetChildrenByTag

Returns a list of children of an element with the given tag name, in the order in which they would be encountered in a preorder traversal of the tree. The tag name should be in the data encoding. The special name "*" matches all tag names; a NULL name matches nothing. Note that tag names are case sensitive. This function is not namespace aware; the full tag names are compared. If two prefixes which map to the same URI are compared, the comparison will fail. See XmlDomGetChildrenByTagNS for the namespace-aware version. The returned list can be freed with XmlDomFreeNodeList.

Syntax

```
xmlnodelist* XmlDomGetChildrenByTag(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	tag name to match; data encoding; * for all

Returns

(xmlnodelist *) node list of matching children

See Also: XmlDomGetChildrenByTagNS, XmlDomFreeNodeList

XmlDomGetChildrenByTagNS

Returns a list of children of an element with the given URI and local name, in the order in which they would be encountered in a preorder traversal of the tree. The URI and local name should be in the data encoding. The special name "*" matches all URIs or tag names; a NULL name matches nothing. Note that names are case sensitive. See XmlDomGetChildrenByTag for the non-namespace version. The returned list can be freed with XmlDomFreeNodeList.

Syntax

```
xmlnodelist* XmlDomGetChildrenByTagNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node

Parameter	In/Out	Description
uri	IN	namespace URI to match; data encoding; * matches all
local	IN	local name to match; data encoding; * matches all

Returns

(xmlnodelist *) node list of matching children

See Also: XmlDomGetChildrenByTag, XmlDomFreeNodeList

XmlDomGetElemsByTag

Returns a list of all elements (in the document tree rooted at the root node) with a given tag name, in the order in which they would be encountered in a preorder traversal of the tree. If root is NULL, the entire document is searched. The tag name should be in the data encoding. The special name "*" matches all tag names; a NULL name matches nothing. Note that tag names are case sensitive. This function is not namespace aware; the full tag names are compared. If two prefixes which map to the same URI are compared, the comparison will fail. See XmlDomGetElemsByTagNS for the namespace-aware version. The returned list can be freed with XmlDomFreeNodeList.

Syntax

```
xmlnodelist* XmlDomGetElemsByTag(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	tag name to match; data encoding; * for all

Returns

(xmlnodelist *) node list of matching elements

See Also: XmlDomGetElemsByTagNS, XmlDomFreeNodeList

XmlDomGetElemsByTagNS

Returns a list of all elements (in the document tree rooted at the root node) with a given URI and localname, in the order in which they would be encountered in a preorder traversal of the tree. If root is `NULL`, the entire document is searched. The tag name should be in the data encoding. The special name "`*`" matches all tag names; a `NULL` name matches nothing. Note that tag names are case sensitive. This function is not namespace aware; the full tag names are compared. If two prefixes which map to the same URI are compared, the comparison will fail. The returned list can be freed with `XmlDomFreeNodeList`.

Syntax

```
xmlnodelist* XmlDomGetElemsByTagNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`elem`	`IN`	element node
`uri`	`IN`	namespace URI to match; data encoding; `*` for all
`local`	`IN`	local name to match; data encoding; `*` for all

Returns

`(xmlnodelist *)` node list of matching elements

See Also: XmlDomGetDocElemsByTag, XmlDomFreeNodeList

XmlDomGetTag

Returns the `tagName`} of a node, which is the same as its name. DOM 1.0 states "...even though there is a generic `nodeName` attribute on the `Node` interface, there is

still a tagName attribute on the Element interface; these two attributes must contain the same value, but the Working Group considers it worthwhile to support both, given the different constituencies the DOM API must satisfy."

Syntax

```
oratext* XmlDomGetTag(
   xmlctx *xctx,
   xmlelemnode *elem)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	Element node

Returns

(oratext *) element's name [data encoding]

See Also: XmlDomGetNodeName

XmlDomHasAttr

Determines if an element has an attribute with the given name. Returns TRUE if so, FALSE if not.

Syntax

```
boolean XmlDomHasAttr(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	Element node
name	IN	attribute's name; data encoding

Returns

`(boolean) TRUE` if element has attribute with given name

See Also: XmlDomHasAttrNS

XmlDomHasAttrNS

Determines if an element has an attribute with the given URI and localname. Returns `TRUE` if so, `FALSE` if not.

Syntax

```
boolean XmlDomHasAttrNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`elem`	`IN`	`Element` node
`uri`	`IN`	attribute's namespace URI; data encoding
`local`	`IN`	attribute's local name; data encoding

Returns

`(boolean) TRUE` if element has attribute with given URI/localname

See Also: XmlDomHasAttr

XmlDomRemoveAttr

Removes an attribute (specified by name). If the removed attribute has a default value it is immediately re-created with that default. Note that the attribute is removed from the element's list of attributes, but the attribute node itself is not destroyed.

Syntax

```
void XmlDomRemoveAttr(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	attribute's name; data encoding

See Also: XmlDomRemoveAttrNS, XmlDomRemoveAttrNode

XmlDomRemoveAttrNS

Removes an attribute (specified by URI and local name). If the removed attribute has a default value it is immediately re-created with that default. Note that the attribute is removed from the element's list of attributes, but the attribute node itself is not destroyed.

Syntax

```
void XmlDomRemoveAttrNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
uri	IN	attribute's namespace URI
local	IN	attribute's local name

See Also: XmlDomRemoveAttr, XmlDomRemoveAttrNode

XmlDomRemoveAttrNode

Removes an attribute from an element. If the attribute has a default value, it is immediately re-created with that value (Specified set to FALSE). Returns the removed attribute on success, else NULL.

Syntax

```
xmlattrnode* XmlDomRemoveAttrNode(
   xmlctx *xctx,
   xmlelemnode *elem,
   xmlattrnode *oldAttr)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
oldAtrr	IN	attribute node to remove

Returns

(xmlattrnode *) replaced attribute node [or NULL]

See Also: XmlDomRemoveAttr

XmlDomSetAttr

Creates a new attribute for an element with the given name and value (which should be in the data encoding). If the named attribute already exists, its value is simply replaced. The name and value are not verified, converted, or checked. The value is not parsed, so entity references will not be expanded. The attribute's specified flag will be set.

Syntax

```
void XmlDomSetAttr(
   xmlctx *xctx,
   xmlelemnode *elem,
```

```
   oratext *name,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
name	IN	attribute's name; data encoding
value	IN	attribute's value; data encoding

See Also: XmlDomSetAttrNS, XmlDomCreateAttr, XmlDomSetAttrValue, XmlDomRemoveAttr

XmlDomSetAttrNS

Creates a new attribute for an element with the given URI, localname and value (which should be in the data encoding). If the named attribute already exists, its value is simply replaced. The name and value are not verified, converted, or checked.

The value is not parsed, so entity references will not be expanded.

The attribute's specified flag will be set.

Syntax

```
void XmlDomSetAttrNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   oratext *uri,
   oratext *qname,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
uri	IN	attribute's namespace URI; data encoding

Parameter	In/Out	Description
qname	IN	attribute's qualified name; data encoding
value	IN	attribute's value; data encoding

See Also: XmlDomSetAttr, XmlDomCreateAttr, XmlDomSetAttrValue, XmlDomRemoveAttr

XmlDomSetAttrNode

Adds a new attribute to an element. If an attribute with the given name already exists, it is replaced and the old attribute returned through oldNode. If the attribute is new, it is added to the element's list and oldNode set to NULL.

Syntax

```
xmlattrnode* XmlDomSetAttrNode(
   xmlctx *xctx,
   xmlelemnode *elem,
   xmlattrnode *newAttr)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
newAttr	IN	attribute node to add

Returns

(xmlattrnode *) replaced attribute node (or NULL)

See Also: XmlDomSetAttrNodeNS, XmlDomCreateAttr, XmlDomSetAttrValue

XmlDomSetAttrNodeNS

Adds a new attribute to an element. If an attribute with newNode's URI and localname already exists, it is replaced and the old attribute returned through

oldNode. If the attribute is new, it is added to the element's list and oldNode set to NULL.

Syntax

```
xmlattrnode* XmlDomSetAttrNodeNS(
   xmlctx *xctx,
   xmlelemnode *elem,
   xmlattrnode *newAttr)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	element node
newAttr	IN	attribute node to add

Returns

(xmlattrnode *) replaced attribute node [or NULL]

See Also: XmlDomSetAttrNode, XmlDomCreateAttr, XmlDomSetAttrValue

Entity Interface

Table 4–6 summarizes the methods of available through the `Entity` interface.

Table 4–6 Summary of Entity Methods; DOM Package

Function	Summary
XmlDomGetEntityNotation on page 4-64	Get entity's notation.
XmlDomGetEntityPubID on page 4-65	Get entity's public ID.
XmlDomGetEntitySysID on page 4-65	Get entity's system ID.
XmlDomGetEntityType on page 4-66	Get entity's type.

XmlDomGetEntityNotation

For unparsed entities, returns the name of its notation (in the data encoding). For parsed entities and other node types, returns `NULL`.

Syntax

```
oratext* XmlDomGetEntityNotation(
   xmlctx *xctx,
   xmlentnode *ent)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`ent`	`IN`	entity node

Returns

`(oratext *)` entity's notation [data encoding; may be `NULL`]

See Also: XmlDomGetEntityPubID, XmlDomGetEntitySysID

XmlDomGetEntityPubID

Returns an entity's public identifier (in the data encoding). If the node is not an entity, or has no defined public ID, returns NULL.

Syntax

```
oratext* XmlDomGetEntityPubID(
   xmlctx *xctx,
   xmlentnode *ent)
```

Parameter	In/Out	Description
xctx	IN	XML context
ent	IN	entity node

Returns

(oratext *) entity's public identifier [data encoding; may be NULL]

See Also: XmlDomGetEntitySysID, XmlDomGetEntityNotation

XmlDomGetEntitySysID

Returns an entity's system identifier (in the data encoding). If the node is not an entity, or has no defined system ID, returns NULL.

Syntax

```
oratext* XmlDomGetEntitySysID(
   xmlctx *xctx,
   xmlentnode *ent)
```

Parameter	In/Out	Description
xctx	IN	XML context
ent	IN	entity node

Returns

`(oratext *)` entity's system identifier [data encoding; may be `NULL`]

See Also: XmlDomGetEntityPubID, XmlDomGetEntityNotation

XmlDomGetEntityType

Returns a boolean for an entity describing whether it is general (`TRUE`) or parameter (`FALSE`).

Syntax

```
boolean XmlDomGetEntityType(
   xmlctx *xctx,
   xmlentnode *ent)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`ent`	`IN`	entity node

Returns

`(boolean)` `TRUE` for general entity, `FALSE` for parameter entity

See Also: XmlDomGetEntityPubID, XmlDomGetEntitySysID, XmlDomGetEntityNotation

NamedNodeMap Interface

Table 4–7 summarizes the methods of available through the `NamedNodeMap` interface.

Table 4–7 Summary of NamedNodeMap Methods; DOM Package

Function	Summary
XmlDomGetNamedItem on page 4-67	Return named node from list.
XmlDomGetNamedItemNS on page 4-68	Return named node from list (namespace aware version).
XmlDomGetNodeMapItem on page 4-69	Return n^{th} node in list.
XmlDomGetNodeMapLength on page 4-69	Return length of named node map.
XmlDomRemoveNamedItem on page 4-70	Remove node from named node map.
XmlDomRemoveNamedItemNS on page 4-71	Remove node from named node map (namespace aware version).
XmlDomSetNamedItem on page 4-71	Set node in named node list.
XmlDomSetNamedItemNS on page 4-72	Set node in named node list (namespace aware version).

XmlDomGetNamedItem

Retrieves an item from a `NamedNodeMap`, specified by name (which should be in the data encoding). This is a non-namespace-aware function; it just matches (case sensitively) on the whole qualified name. Note this function differs from the DOM spec in that the index of the matching item is also returned.

Syntax

```
xmlnode* XmlDomGetNamedItem(
   xmlctx *xctx,
   xmlnamedmap *map,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap
name	IN	name of the node to retrieve

Returns

(xmlnode *) Node with the specified name [or NULL]

See Also: XmlDomGetNamedItemNS, XmlDomGetNodeMapItem, XmlDomGetNodeMapLength

XmlDomGetNamedItemNS

Retrieves an item from a NamedNodeMap, specified by URI and localname (which should be in the data encoding). Note this function differs from the DOM spec in that the index of the matching item is also returned.

Syntax

```
xmlnode* XmlDomGetNamedItemNS(
   xmlctx *xctx,
   xmlnamedmap *map,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap
uri	IN	namespace URI of the node to retrieve; data encoding
local	IN	local name of the node to retrieve; data encoding

Returns

(xmlnode *) node with given local name and namespace URI [or NULL]

See Also: XmlDomGetNamedItem, XmlDomGetNodeMapItem, XmlDomGetNodeMapLength

XmlDomGetNodeMapItem

Retrieves an item from a `NamedNodeMap`, specified by name (which should be in the data encoding). This is a non-namespace-aware function; it just matches (case sensitively) on the whole qualified name. Note this function differs from the DOM specification in that the index of the matching item is also returned. Named "item" in W3C specification.

Syntax

```
xmlnode* XmlDomGetNodeMapItem(
   xmlctx *xctx,
   xmlnamedmap *map,
   ub4 index)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`map`	`IN`	`NamedNodeMap`
`index`	`IN`	`0`-based index for the map

Returns

`(xmlnode *)` node at the nth position in the map (or `NULL`)

See Also: XmlDomGetNamedItem, XmlDomSetNamedItem, XmlDomRemoveNamedItem, XmlDomGetNodeMapLength

XmlDomGetNodeMapLength

Returns the number of nodes in a `NamedNodeMap` (the length). Note that nodes are referred to by index, and the range of valid indexes is `0` through length-1.

Syntax

```
ub4 XmlDomGetNodeMapLength(
```

```
   xmlctx *xctx,
   xmlnamedmap *map)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap

Returns

(ub4) number of nodes in NamedNodeMap

See Also: XmlDomGetNodeMapItem, XmlDomGetNamedItem

XmlDomRemoveNamedItem

Removes a node from a NamedNodeMap, specified by name. This is a non-namespace-aware function; it just matches (case sensitively) on the whole qualified name. If the removed node is an attribute with default value (not specified), it is immediately replaced. The removed node is returned; if no removal took place, NULL is returned.

Syntax

```
xmlnode* XmlDomRemoveNamedItem(
   xmlctx *xctx,
   xmlnamedmap *map,
   oratext *name)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap
name	IN	name of node to remove

Returns

(xmlnode *) node removed from this map

See Also: XmlDomRemoveNamedItemNS, XmlDomGetNamedItem, XmlDomGetNamedItemNS, XmlDomSetNamedItem, XmlDomSetNamedItemNS

XmlDomRemoveNamedItemNS

Removes a node from a NamedNodeMap, specified by URI and localname. If the removed node is an attribute with default value (not specified), it is immediately replaced. The removed node is returned; if no removal took place, NULL is returned.

Syntax

```
xmlnode* XmlDomRemoveNamedItemNS(
   xmlctx *xctx,
   xmlnamedmap *map,
   oratext *uri,
   oratext *local)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap
uri	IN	namespace URI of the node to remove; data encoding
local	IN	local name of the node to remove; data encoding

Returns

(xmlnode *) node removed from this map

See Also: XmlDomRemoveNamedItem, XmlDomGetNamedItem, XmlDomGetNamedItemNS, XmlDomSetNamedItem, XmlDomSetNamedItemNS

XmlDomSetNamedItem

Adds a new node to a NamedNodeMap. If a node already exists with the given name, replaces the old node and returns it. If no such named node exists, adds the new node to the map and sets old to NULL. This is a non-namespace-aware function;

it just matches (case sensitively) on the whole qualified name. Since some node types have fixed names (`Text`, `Comment`, and so on), trying to set another of the same type will always cause replacement.

Syntax

```
xmlnode* XmlDomSetNamedItem(
   xmlctx *xctx,
   xmlnamedmap *map,
   xmlnode *newNode)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`map`	`IN`	`NamedNodeMap`
`newNode`	`IN`	new node to store in map

Returns

`(xmlnode *)` the replaced node (or `NULL`)

See Also: XmlDomSetNamedItemNS, XmlDomGetNamedItem, XmlDomGetNamedItemNS, XmlDomGetNodeMapItem, XmlDomGetNodeMapLength

XmlDomSetNamedItemNS

Adds a new node to a `NamedNodeMap`. If a node already exists with the given URI and localname, replaces the old node and returns it. If no such named node exists, adds the new node to the map and sets old to `NULL`. Since some node types have fixed names (`Text`, `Comment`, and so on), trying to set another of the same type will always cause replacement.

Syntax

```
xmlnode* XmlDomSetNamedItemNS(
   xmlctx *xctx,
   xmlnamedmap *map,
   xmlnode *newNode)
```

Parameter	In/Out	Description
xctx	IN	XML context
map	IN	NamedNodeMap
newNode	IN	new node to store in map

Returns

(xmlnode *) replaced Node [or NULL]

See Also: XmlDomSetNamedItem, XmlDomGetNamedItem, XmlDomGetNamedItemNS, XmlDomGetNodeMapItem, XmlDomGetNodeMapLength

Node Interface

Table 4–8 summarizes the methods of available through the `Node` interface.

Table 4–8 Summary of Text Methods; DOM Package

Function	Summary
XmlDomAppendChild on page 4-76	Append new child to node's list of children.
XmlDomCleanNode on page 4-76	Clean a node (free DOM allocations).
XmlDomCloneNode on page 4-77	Clone a node.
XmlDomFreeNode on page 4-78	Free a node allocated with `XmlDomCreateXXX`.
XmlDomGetAttrs on page 4-78	Return attributes of node.
XmlDomGetChildNodes on page 4-79	Return children of node.
XmlDomGetDefaultNS on page 4-80	Get default namespace for node.
XmlDomGetFirstChild on page 4-80	Returns first child of node.
XmlDomGetFirstPfnsPair on page 4-81	Get first prefix namespace pair.
XmlDomGetLastChild on page 4-81	Returns last child of node.
XmlDomGetNextPfnsPair on page 4-82	Get subsequent prefix namespace pair.
XmlDomGetNextSibling on page 4-83	Return next sibling of node.
XmlDomGetNodeLocal on page 4-83	Get local part of node's qualified name as `NULL`-terminated string.
XmlDomGetNodeLocalLen on page 4-84	Get local part of node's qualified name as length-encoded string.
XmlDomGetNodeName on page 4-85	Get node's name as `NULL`-terminated string.
XmlDomGetNodeNameLen on page 4-86	Get node's name as length-encoded string.
XmlDomGetNodePrefix on page 4-87	Return namespace prefix of node.
XmlDomGetNodeType on page 4-87	Get node's numeric type code.
XmlDomGetNodeURI on page 4-89	Return namespace URI of node as a `NULL`-terminated string.
XmlDomGetNodeURILen on page 4-89	Return namespace URI of node as length-encoded string.

Table 4–8 (Cont.) Summary of Text Methods; DOM Package

Function	Summary
XmlDomGetNodeValue on page 4-90	Get node's value as NULL-terminated string.
XmlDomGetNodeValueLen on page 4-91	Get node value as length-encoded string.
XmlDomGetNodeValueStream on page 4-92	Get node value stream-style (chunked).
XmlDomGetOwnerDocument on page 4-93	Get the owner document of node.
XmlDomGetParentNode on page 4-93	Get parent node.
XmlDomGetPrevSibling on page 4-94	Return previous sibling of node.
XmlDomGetSourceEntity on page 4-95	Return the entity node if the input file is an external entity.
XmlDomGetSourceLine on page 4-95	Return source line number of node.
XmlDomGetSourceLocation on page 4-96	Return source location (path, URI, and so on) of node.
XmlDomHasAttr on page 4-57	Does named attribute exist?
XmlDomHasChildNodes on page 4-97	Test if node has children.
XmlDomInsertBefore on page 4-97	Insert new child in to node's list of children.
XmlDomNormalize on page 4-98	Normalize a node by merging adjacent text nodes.
XmlDomNumAttrs on page 4-98	Return number of attributes of element.
XmlDomNumChildNodes on page 4-99	Return number of children of node.
XmlDomPrefixToURI on page 4-99	Get namespace URI for prefix.
XmlDomRemoveChild on page 4-100	Remove an existing child node.
XmlDomReplaceChild on page 4-101	Replace an existing child of a node.
XmlDomSetDefaultNS on page 4-101	Set default namespace for node.
XmlDomSetNodePrefix on page 4-102	Set namespace prefix of node.
XmlDomSetNodeValue on page 4-102	Set node value.
XmlDomSetNodeValueLen on page 4-103	Set node value as length-encoded string.
XmlDomSetNodeValueStream on page 4-104	Set node value stream-style (chunked).
XmlDomValidate on page 4-105	Validate a node against current DTD.

XmlDomAppendChild

Appends the node to the end of the parent's list of children and returns the new node. If `newChild` is a `DocumentFragment`, all of its children are appended in original order; the `DocumentFragment` node itself is not.

Syntax

```
xmlnode* XmlDomAppendChild(
   xmlctx *xctx,
   xmlnode *parent,
   xmlnode *newChild)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`parent`	`IN`	parent to receive a new node
`newChild`	`IN`	node to add

Returns

`(xmlnode *)` node added

See Also: XmlDomInsertBefore, XmlDomReplaceChild

XmlDomCleanNode

Frees parts of the node which were allocated by DOM itself, but does not recurse to children or touch the node's attributes. After freeing part of the node (such as name), a DOM call to get that part (such as `XmlDomGetNodeName`) should return a `NULL` pointer. Used to manage the allocations of a node parts of which are controlled by DOM, and part by the user. Calling clean frees all allocations may by DOM and leaves the user's allocations alone. The user is responsible for freeing their own allocations.

Syntax

```
void XmlDomCleanNode(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	node to clean

See Also: XmlDomFreeNode

XmlDomCloneNode

Creates and returns a duplicate of a node. The duplicate node has no parent. Cloning an element copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but it does not copy any text it contains unless it is a deep clone, since the text is contained in a child text node. Cloning any other type of node simply returns a copy of the node. Note that a clone of an unspecified attribute node is specified. If deep is TRUE, all children of the node are recursively cloned, and the cloned node will have cloned children; a non-deep clone will have no children.

Syntax

```
xmlnode* XmlDomCloneNode(
   xmlctx *xctx,
   xmlnode *node,
   boolean deep)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
deep	IN	TRUE to recursively clone children

Returns

(xmlnode *) duplicate (cloned) node

See Also: XmlDomImportNode

XmlDomFreeNode

Free a node allocated with `XmlDomCreate`*XXX*. Frees all resources associated with a node, then frees the node itself. Certain parts of the node are under DOM control, and some parts may be under user control. DOM keeps flags tracking who owns what, and only frees its own allocations. The user is responsible for freeing their own parts of the node before calling XmlDomFreeNode.

Syntax

```
void XmlDomFreeNode(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`node`	`IN`	XML node to free

See Also: XmlDomCleanNode

XmlDomGetAttrs

Returns a `NamedNodeMap` of attributes of an element node, or `NULL` if it has no attributes. For other node types, `NULL` is returned. Note that if an element once had attributes, but they have all been removed, an empty list will be returned. So, presence of the list does not mean the element has attributes. You must check the size of the list with `XmlDomNumAttrs` or use `XmlDomHasChildNodes` first.

Syntax

```
xmlnamedmap* XmlDomGetAttrs(
   xmlctx *xctx,
   xmlelemnode *elem)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context

Parameter	In/Out	Description
elem	IN	XML element node

Returns

(xmlnamedmap *) NamedNodeMap of node's attributes

See Also: XmlDomNumAttrs, XmlDomHasChildNodes

XmlDomGetChildNodes

Returns a list of the node's children, or NULL if it has no children. Only Element, Document, DTD, and DocumentFragment nodes may have children; all other types will return NULL.

Note that an empty list may be returned if the node once had children, but all have been removed! That is, the list may exist but have no members. So, presence of the list alone does not mean the node has children. You must check the size of the list with XmlDomNumChildNodes or use XmlDomHasChildNodes first.

The xmlnodelist structure is opaque and can only be manipulated with functions in the NodeList interface.

The returned list is live; all changes in the original node are reflected immediately.

Syntax

```
xmlnodelist* XmlDomGetChildNodes(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnodelist *) live NodeList containing all children of node

XmlDomGetDefaultNS

Gets the default namespace for a node.

Syntax

```
oratext* XmlDomGetDefaultNS(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	element or attribute DOM node

Returns

(oratext *) default namespace for node [data encoding; may be NULL]

XmlDomGetFirstChild

Returns the first child of a node, or NULL if the node has no children. Only Element, Document, DTD, and DocumentFragment nodes may have children; all other types will return NULL.

Syntax

```
xmlnode* XmlDomGetFirstChild(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnode *) first child of node

See Also: XmlDomGetLastChild, XmlDomHasChildNodes, XmlDomGetChildNodes, XmlDomNumChildNodes

XmlDomGetFirstPfnsPair

This function is to allow implementations an opportunity to speedup the iteration of all available prefix-URI bindings available on a given node. It returns a state structure and the prefix and URI of the first prefix-URI mapping. The state structure should be passed to `XmlDomGetNextPfnsPair` on the remaining pairs.

Syntax

```
xmlpfnspair* XmlDomGetFirstPfnsPair(
   xmlctx *xctx,
   xmlnode *node,
   oratext **prefix,
   oratext **uri)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`node`	`IN`	XML node
`prefix`	`OUT`	prefix of first mapping; data encoding
`uri`	`OUT`	URI of first mapping; data encoding

Returns

`(xmlpfnspair *)` iterating object or `NULL` of no prefixes

XmlDomGetLastChild

Returns the last child of a node, or `NULL` if the node has no children. Only `Element`, `Document`, DTD, and `DocumentFragment` nodes may have children; all other types will return `NULL`.

Syntax

```
xmlnode* XmlDomGetLastChild(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnode *) last child of node

See Also: XmlDomGetFirstChild, XmlDomHasChildNodes, XmlDomGetChildNodes, XmlDomNumChildNodes

XmlDomGetNextPfnsPair

This function is to allow implementations an opportunity to speedup the iteration of all available prefix-URI bindings available on a given node. Given an iterator structure from XmlDomGetFirstPfnsPair, returns the next prefix-URI mapping; repeat calls to XmlDomGetNextPfnsPair until NULL is returned.

Syntax

```
xmlpfnspair* XmlDomGetNextPfnsPair(
   xmlctx *xctx
   xmlpfnspair *pfns,
   oratext **prefix,
   oratext **uri)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
prefix	OUT	prefix of next mapping; data encoding
uri	OUT	URI of next mapping; data encoding

Returns

(xmlpfnspair *) iterating object, NULL when no more pairs

XmlDomGetNextSibling

Returns the node following a node at the same level in the DOM tree. That is, for each child of a parent node, the next sibling of that child is the child which comes after it. If a node is the last child of its parent, or has no parent, NULL is returned.

Syntax

```
xmlnode* XmlDomGetNextSibling(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnode *) node immediately following node at same level

See Also: XmlDomGetPrevSibling

XmlDomGetNodeLocal

Returns the namespace local name for a node as a NULL-terminated string. If the node's name is not fully qualified (has no prefix), then the local name is the same as the name.

A length-encoded version is available as XmlDomGetNodeLocalLen which returns the local name as a pointer and length, for use if the data is known to use XMLType backing store.

Syntax

```
oratext* XmlDomGetNodeLocal(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(oratext *) local name of node [data encoding]

See Also: XmlDomGetNodeLocalLen, XmlDomGetNodePrefix, XmlDomGetNodeURI

XmlDomGetNodeLocalLen

Returns the namespace local name for a node as a length-encoded string. If the node's name is not fully qualified (has no prefix), then the local name is the same as the name.

A NULL-terminated version is available as XmlDomGetNodeLocal which returns the local name as NULL-terminated string. If the backing store is known to be XMLType, then the node's data will be stored internally as length-encoded. Using the length-based Get functions will avoid having to copy and NULL-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetNodeLocalLen(
   xmlctx *xctx,
   xmlnode *node,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of local name, in characters

Returns

(oratext *) local name of node [data encoding]

See Also: XmlDomGetNodeLocal, XmlDomGetNodePrefix, XmlDomGetNodeURILen

XmlDomGetNodeName

Returns the (fully-qualified) name of a node (in the data encoding) as a NULL-terminated string, for example bar\0 or foo:bar\0.

Note that some node types have fixed names: "#text", "#cdata-section", "#comment", "#document", "#document-fragment".

A node's name cannot be changed once it is created, so there is no matching SetNodeName function.

A length-based version is available as XmlDomGetNodeNameLen which returns the node name as a pointer and length, for use if the data is known to use XMLType backing store.

Syntax

```
oratext* XmlDomGetNodeName(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

`(oratext *)` name of node [data encoding]

See Also: XmlDomGetNodeNameLen

XmlDomGetNodeNameLen

Returns the (fully-qualified) name of a node (in the data encoding) as a length-encoded string, for example `"bar", 3` or `"foo:bar", 7`.

Note that some node types have fixed names: `"#text"`, `"#cdata-section"`, `"#comment"`, `"#document"`, `"#document-fragment"`.

A node's name cannot be changed once it is created, so there is no matching SetNodeName function.

A `NULL`-terminated version is available as XmlDomGetNodeName which returns the node name as `NULL`-terminated string. If the backing store is known to be `XMLType`, then the node's name will be stored internally as length-encoded. Using the length-encoded `Get`*`XXX`* functions will avoid having to copy and `NULL`-terminate the name.

If both the input buffer is non-`NULL` and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and `len` will return the actual length.

Syntax

```
oratext* XmlDomGetNodeNameLen(
   xmlctx *xctx,
   xmlnode *node,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Parameter	In/Out	Description
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of name, in characters

Returns

(oratext *) name of node, with length of name set in 'len'

See Also: XmlDomGetNodeName

XmlDomGetNodePrefix

Returns the namespace prefix for a node (as a NULL-terminated string). If the node's name is not fully qualified (has no prefix), NULL is returned.

Syntax

```
oratext* XmlDomGetNodePrefix(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(oratext *) namespace prefix of node [data encoding; may be NULL]

XmlDomGetNodeType

Returns the type code of a node. The type names and numeric values match the DOM specification:

- ELEMENT_NODE=1
- ATTRIBUTE_NODE=2

- TEXT_NODE=3
- CDATA_SECTION_NODE=4
- ENTITY_REFERENCE_NODE=5
- ENTITY_NODE=6
- PROCESSING_INSTRUCTION_NODE=7
- COMMENT_NODE=8
- DOCUMENT_NODE=9
- DOCUMENT_TYPE_NODE=10
- DOCUMENT_FRAGMENT_NODE=11
- NOTATION_NODE=12

Additional Oracle extension node types are as follows:

- ELEMENT_DECL_NODE
- ATTR_DECL_NODE
- CP_ELEMENT_NODE
- CP_CHOICE_NODE
- CP_PCDATA_NODE
- CP_STAR_NODE
- CP_PLUS_NODE
- CP_OPT_NODE

Syntax

```
xmlnodetype XmlDomGetNodeType(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

`(xmlnodetype)` numeric type-code of the node

XmlDomGetNodeURI

Returns the namespace URI for a node (in the data encoding) as a `NULL`-terminated string. If the node's name is not qualified (does not contain a namespace prefix), it will have the default namespace in effect when the node was created (which may be `NULL`).

A length-encoded version is available as `XmlDomGetNodeURILen` which returns the URI as a pointer and length, for use if the data is known to use `XMLType` backing store.

Syntax

```
oratext* XmlDomGetNodeURI(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`node`	`IN`	XML node

Returns

`(oratext *)` namespace URI of node [data encoding; may be `NULL`]

See Also: XmlDomGetNodeURILen, XmlDomGetNodePrefix, XmlDomGetNodeLocal

XmlDomGetNodeURILen

Returns the namespace URI for a node (in the data encoding) as length-encoded string. If the node's name is not qualified (does not contain a namespace prefix), it will have the default namespace in effect when the node was created (which may be `NULL`).

A `NULL`-terminated version is available as `XmlDomGetNodeURI` which returns the URI value as `NULL`-terminated string. If the backing store is known to be `XMLType`,

then the node's data will be stored internally as length-encoded. Using the length-based Get functions will avoid having to copy and NULL-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetNodeURILen(
   xmlctx *xctx,
   xmlnode *node,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of URI, in characters

Returns

(oratext *) namespace URI of node [data encoding; may be NULL]

See Also: XmlDomGetNodeURI, XmlDomGetNodePrefix, XmlDomGetNodeLocal

XmlDomGetNodeValue

Returns the "value" (associated character data) for a node as a NULL-terminated string. Character and general entities will have been replaced. Only Attr, CDATA, Comment, ProcessingInstruction and Text nodes have values, all other node types have NULL value.

A length-encoded version is available as XmlDomGetNodeValueLen which returns the node value as a pointer and length, for use if the data is known to use XMLType backing store.

Syntax

```
oratext* XmlDomGetNodeValue(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(oratext *) value of node

See Also: XmlDomSetNodeValue, XmlDomGetNodeValueLen

XmlDomGetNodeValueLen

Returns the "value" (associated character data) for a node as a length-encoded string. Character and general entities will have been replaced. Only Attr, CDATA, Comment, PI and Text nodes have values, all other node types have NULL value.

A NULL-terminated version is available as XmlDomGetNodeValue which returns the node value as NULL-terminated string. If the backing store is known to be XMLType, then the node's data will be stored internally as length-encoded. Using the length-based Get functions will avoid having to copy and NULL-terminate the data.

If both the input buffer is non-NULL and the input buffer length is nonzero, then the value will be stored in the input buffer. Else, the implementation will return its own buffer.

If the actual length is greater than buflen, then a truncated value will be copied into the buffer and len will return the actual length.

Syntax

```
oratext* XmlDomGetNodeValueLen(
   xmlctx *xctx,
   xmlnode *node,
   oratext *buf,
   ub4 buflen,
   ub4 *len)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
buf	IN	input buffer; optional
buflen	IN	input buffer length; optional
len	OUT	length of value, in bytes

Returns

`(oratext *)` value of node

See Also: XmlDomSetNodeValueLen, XmlDomGetNodeValue

XmlDomGetNodeValueStream

Returns the large data for a node and sends it in pieces to the user's output stream. For very large values, it is not always possible to store them [efficiently] as a single contiguous chunk. This function is used to access chunked data of that type. Only `XMLType` chunks its data (sometimes); XDK's data is always contiguous.

Syntax

```
xmlerr XmlDomGetNodeValueStream(
   xmlctx *xctx,
   xmlnode *node,
   xmlostream *ostream)
```

Parameter	In/Out	Description
xctx	IN	XML context

Parameter	In/Out	Description
node	IN	XML node
ostream	IN	output stream object

Returns

(xmlerr) numeric error code, 0 on success

See Also: XmlDomSetNodeValueStream, XmlDomGetNodeValue, XmlDomGetNodeValueLen

XmlDomGetOwnerDocument

Returns the Document node associated with a node. Each node may belong to only one document, or may not be associated with any document at all (such as immediately after XmlDomCreateElem, and so on). The "owning" document [node] is returned, or NULL for an orphan node.

Syntax

```
xmldocnode* XmlDomGetOwnerDocument(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmldocnode *) document node is in

XmlDomGetParentNode

Returns a node's parent node. All nodes types except Attr, Document, DocumentFragment, Entity, and Notation may have a parent (these five exceptions always have a NULL parent). If a node has just been created but not yet

added to the DOM tree, or if it has been removed from the DOM tree, its parent is also NULL.

Syntax

```
xmlnode* XmlDomGetParentNode(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnode *) parent of node

XmlDomGetPrevSibling

Returns the node preceding a node at the same level in the DOM tree. That is, for each child of a parent node, the previous sibling of that child is the child which came before it. If a node is the first child of its parent, or has no parent, NULL is returned.

Syntax

```
xmlnode* XmlDomGetPrevSibling(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlnode *) node immediately preceding node at same level

See Also: XmlDomGetNextSibling

XmlDomGetSourceEntity

Returns the external entity node whose inclusion caused the creation of the given node.

Syntax

```
xmlentnode* XmlDomGetSourceEntity(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(xmlentnode *) entity node if the input is from an external entity

XmlDomGetSourceLine

Returns the line# in the original source where the node started. The first line in every input is line #1.

Syntax

```
ub4 XmlDomGetSourceLine(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(ub4) line number of node in original input source

XmlDomGetSourceLocation

Return source location (path, URI, and so on) of node. Note this will be in the compiler encoding, not the data encoding!

Syntax

```
oratext* XmlDomGetSourceLocation(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

`(oratext *)` full path of input source [in compiler encoding]

XmlDomHasAttrs

Test if an element has attributes. Returns `TRUE` if any attributes of any sort are defined (namespace or regular).

Syntax

```
boolean XmlDomHasAttrs(
   xmlctx *xctx,
   xmlelemnode *elem)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	XML element node

Returns

`(boolean) TRUE` if element has attributes

XmlDomHasChildNodes

Test if a node has children. Only `Element`, `Document`, DTD, and `DocumentFragment` nodes may have children. Note that just because `XmlDomGetChildNodes` returns a list does not mean the node actually has children, since the list may be empty, so a non-`NULL` return from `XmlDomGetChildNodes` should not be used as a test.

Syntax

```
boolean XmlDomHasChildNodes(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`node`	`IN`	XML node

Returns

`(boolean) TRUE` if the node has any children

XmlDomInsertBefore

Inserts the node `newChild` before the existing child node `refChild` in the parent node. If `refChild` is `NULL`, appends to parent's children as for each `XmlDomAppendChild`; otherwise it must be a child of the given parent. If `newChild` is a `DocumentFragment`, all of its children are inserted (in the same order) before `refChild`; the `DocumentFragment` node itself is not. If `newChild` is already in the DOM tree, it is first removed from its current position.

Syntax

```
xmlnode* XmlDomInsertBefore(
   xmlctx *xctx,
   xmlnode *parent,
   xmlnode *newChild,
   xmlnode *refChild)
```

Parameter	In/Out	Description
xctx	IN	XML context
parent	IN	parent that receives a new child
newChild	IN	node to insert
refChild	IN	reference node

Returns

(xmlnode *) node being inserted

See Also: XmlDomAppendChild, XmlDomReplaceChild, XmlDomRemoveChild

XmlDomNormalize

Normalizes the subtree rooted at an element, merges adjacent Text nodes children of elements. Note that adjacent Text nodes will never be created during a normal parse, only after manipulation of the document with DOM calls.

Syntax

```
void XmlDomNormalize(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

XmlDomNumAttrs

Returns the number of attributes of an element. Note that just because a list is returned by XmlDomGetAttrs does not mean it contains any attributes; it may be an empty list with zero length.

Syntax

```
ub4 XmlDomNumAttrs(
   xmlctx *xctx,
   xmlelemnode *elem)
```

Parameter	In/Out	Description
xctx	IN	XML context
elem	IN	XML element node

Returns

(ub4) number of attributes of node

XmlDomNumChildNodes

Returns the number of children of a node. Only Element, Document, DTD, and DocumentFragment nodes may have children, all other types return 0. Note that just because XmlDomGetChildNodes returns a list does not mean that it contains any children; it may be an empty list with zero length.

Syntax

```
ub4 XmlDomNumChildNodes(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node

Returns

(ub4) number of children of node

XmlDomPrefixToURI

Given a namespace prefix and a node, returns the namespace URI mapped to that prefix. If the given node doesn't have a matching prefix, its parent is tried, then its

parent, and so on, all the way to the root node. If the prefix is undefined, NULL is returned.

Syntax

```
oratext* XmlDomPrefixToURI(
   xmlctx *xctx,
   xmlnode *node,
   oratext *prefix)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
prefix	IN	prefix to map

Returns

(oratext *) URI for prefix [data encoding; NULL if no match]

XmlDomRemoveChild

Removes a node from its parent's list of children and returns it. The node is orphaned; its parent will be NULL after removal.

Syntax

```
xmlnode* XmlDomRemoveChild(
   xmlctx *xctx,
   xmlnode *oldChild)
```

Parameter	In/Out	Description
xctx	IN	XML context
oldChild	IN	node to remove

Returns

(xmlnode *) node removed

See Also: XmlDomAppendChild, XmlDomInsertBefore, XmlDomReplaceChild

XmlDomReplaceChild

Replaces the child node `oldChild` with the new node `newChild` in `oldChild`'s parent, and returns `oldChild` (which is now orphaned, with a `NULL` parent). If `newChild` is a `DocumentFragment`, all of its children are inserted in place of `oldChild`; the `DocumentFragment` node itself is not. If `newChild` is already in the DOM tree, it is first removed from its current position.

Syntax

```
xmlnode* XmlDomReplaceChild(
   xmlctx *xctx,
   xmlnode *newChild,
   xmlnode *oldChild)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`newChild`	`IN`	new node that is substituted
`oldChild`	`IN`	old node that is replaced

Returns

`(xmlnode *)` node replaced

See Also: XmlDomAppendChild, XmlDomInsertBefore, XmlDomRemoveChild

XmlDomSetDefaultNS

Set the default namespace for a node

Syntax

```
void XmlDomSetDefaultNS(
   xmlctx *xctx,
```

```
   xmlnode *node,
   oratext *defns)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	element or attribute DOM node
defns	IN	new default namespace for the node

XmlDomSetNodePrefix

Sets the namespace prefix of node (as NULL-terminated string). Does not verify the prefix is defined. Just causes a new qualified name to be formed from the new prefix and the old local name; the new qualified name will be under DOM control and should not be managed by the user.

Syntax

```
void XmlDomSetNodePrefix(
   xmlctx *xctx,
   xmlnode *node,
   oratext *prefix)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
prefix	OUT	new namespace prefix

XmlDomSetNodeValue

Sets a node's value (character data) as a NULL-terminated string. Does not allow setting the value to NULL. Only Attr, CDATA, Comment, PI and Text nodes have values; trying to set the value of another type of node is a no-op. The new value must be in the data encoding. It is not verified, converted, or checked.

The value is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
xmlerr XmlDomSetNodeValue(
   xmlctx *xctx,
   xmlnode *node,
   oratext *value)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
value	IN	node's new value; data encoding; user control

Returns

(xmlerr) numeric error code, 0 on success

See Also: XmlDomGetNodeValue, XmlDomSetNodeValueLen

XmlDomSetNodeValueLen

Sets the value (associated character data) for a node as a length-encoded string.

A NULL-terminated version is available as XmlDomSetNodeValue which takes the node value as a NULL-terminated string. If the backing store is known to be XMLType, then the node's data will be stored internally as length-encoded. Using the length-based Set functions will avoid having to copy and NULL-terminate the data.

Syntax

```
xmlerr XmlDomSetNodeValueLen(
   xmlctx *xctx,
   xmlnode *node,
   oratext *value,
   ub4 len)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
value	IN	node's new value; data encoding; user control
len	IN	length of value, in bytes

Returns

(xmlerr) numeric error code, 0 on success

See Also: XmlDomSetNodeValueLen, XmlDomSetNodeValue

XmlDomSetNodeValueStream

Sets the large "value" (character data) for a node piecemeal from an input stream. For very large values, it is not always possible to store them [efficiently] as a single contiguous chunk. This function is used to store chunked data of that type. Used only for XMLType data; XDK's data is always contiguous.

Syntax

```
xmlerr XmlDomSetNodeValueStream(
   xmlctx *xctx,
   xmlnode *node,
   xmlistream *istream)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	XML node
istream	IN	input stream object

Returns

(xmlerr) numeric error code, 0 on success

See Also: XmlDomGetNodeValueStream, XmlDomSetNodeValue

XmlDomValidate

Given a root node, validates it against the current DTD.

Syntax

```
xmlerr XmlDomValidate(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context
node	IN	node to validate

Returns

(xmlerr) error code, XMLERR_OK [0] means node is valid

NodeList Interface

Table 4–9 summarizes the methods of available through the `NodeList` interface.

Table 4–9 Summary of NodeList Methods; DOM Package

Function	Summary
XmlDomFreeNodeList on page 4-106	Free a node list returned by XmlDomGetElemsByTag, and so on.
XmlDomGetNodeListItem on page 4-106	Return n^{th} node in list.
XmlDomGetNodeListLength on page 4-107	Return length of node list.

XmlDomFreeNodeList

Free a node list returned by `XmlDomGetElemsByTag` or related functions, releasing all resources associated with it. If given a node list that is part of the DOM proper (such as the children of a node), does nothing.

Syntax

```
void XmlDomFreeNodeList(
   xmlctx *xctx,
   xmlnodelist *list)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`list`	`IN`	`NodeList` to free

See Also: XmlDomGetElemsByTag, XmlDomGetElemsByTagNS, XmlDomGetChildrenByTag, XmlDomGetChildrenByTagNS

XmlDomGetNodeListItem

Return nth node in a node list. The first item is index `0`.

Syntax

```
xmlnode* XmlDomGetNodeListItem(
   xmlctx *xctx,
   xmlnodelist *list,
   ub4 index)
```

Parameter	In/Out	Description
xctx	IN	XML context
list	IN	NodeList
index	IN	index into list

Returns

(xmlnode *) node at the nth position in node list [or NULL]

See Also: XmlDomGetNodeListLength, XmlDomFreeNodeList

XmlDomGetNodeListLength

Returns the number of nodes in a node list (its length). Note that nodes are referred to by index, so the range of valid indexes is 0 through length-1.

Syntax

```
ub4 XmlDomGetNodeListLength(
   xmlctx *xctx,
   xmlnodelist *list)
```

Parameter	In/Out	Description
xctx	IN	XML context
list	IN	NodeList

Returns

(ub4) number of nodes in node list

See Also: XmlDomGetNodeListItem, XmlDomFreeNodeList

Notation Interface

Table 4–10 summarizes the methods of available through the `Notation` interface.

Table 4–10 Summary of NodeList Methods; DOM Package

Function	Summary
XmlDomGetNotationPubID on page 4-109	Get notation's public ID
XmlDomGetNotationSysID on page 4-109	Get notation's system ID.

XmlDomGetNotationPubID

Return a notation's public identifier (in the data encoding). If the node is not a notation, or has no defined public ID, returns `NULL`.

Syntax

```
oratext* XmlDomGetNotationPubID(
   xmlctx *xctx,
   xmlnotenode *note)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`note`	`IN`	`Notation` node

Returns

`(oratext *)` notation's public identifier [data encoding; may be `NULL`]

See Also: XmlDomGetNotationSysID

XmlDomGetNotationSysID

Return a notation's system identifier (in the data encoding). If the node is not a notation, or has no defined system ID, returns `NULL`.

Syntax

```
oratext* XmlDomGetNotationSysID(
   xmlctx *xctx,
   xmlnotenode *note)
```

Parameter	In/Out	Description
xctx	IN	XML context
note	IN	Notation node

Returns

(oratext *) notation's system identifier [data encoding; may be NULL]

See Also: XmlDomGetNotationPubID

ProcessingInstruction Interface

Table 4–11 summarizes the methods of available through the `ProcessingInstruction` interface.

Table 4–11 Summary of ProcessingInstruction Methods; DOM Package

Function	Summary
XmlDomGetPIData on page 4-111	Get processing instruction's data.
XmlDomGetPITarget on page 4-112	Get PI's target.
XmlDomSetPIData on page 4-112	Set processing instruction's data.

XmlDomGetPIData

Returns the content (data) of a processing instruction (in the data encoding). If the node is not a `ProcessingInstruction`, returns `NULL`. The content is the part from the first non-whitespace character after the target until the ending "`?>`".

Syntax

```
oratext* XmlDomGetPIData(
   xmlctx *xctx,
   xmlpinode *pi)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`pi`	`IN`	`ProcessingInstruction` node

Returns

`(oratext *)` processing instruction's data [data encoding]

See Also: XmlDomGetPITarget, XmlDomSetPIData

XmlDomGetPITarget

Returns a processing instruction's target string. If the node is not a `ProcessingInstruction`, returns `NULL`. The target is the first token following the markup that begins the `ProcessingInstruction`. All `ProcessingInstructions` must have a target, though the data part is optional.

Syntax

```
oratext* XmlDomGetPITarget(
   xmlctx *xctx,
   xmlpinode *pi)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`pi`	`IN`	`ProcessingInstruction` node

Returns

`(oratext *)` processing instruction's target [data encoding]

See Also: XmlDomGetPIData, XmlDomSetPIData

XmlDomSetPIData

Sets a `ProcessingInstruction`'s content, which must be in the data encoding. It is not permitted to set the data to `NULL`. If the node is not a `ProcessingInstruction`, does nothing. The new data is not verified, converted, or checked.

Syntax

```
void XmlDomSetPIData(
   xmlctx *xctx,
   xmlpinode *pi,
   oratext *data)
```

Parameter	In/Out	Description
xctx	IN	XML context
pi	IN	ProcessingInstruction node
data	IN	ProcessingInstruction's new data; data encoding

See Also: XmlDomGetPITarget, XmlDomGetPIData

Text Interface

Table 4–12 summarizes the methods of available through the `Text` interface.

Table 4–12 Summary of Text Methods; DOM Package

Function	Summary
XmlDomSplitText on page 4-114	Split text node in to two.

XmlDomSplitText

Splits a single text node into two text nodes; the original data is split between them. If the given node is not type text, or the offset is outside of the original data, does nothing and returns `NULL`. The offset is zero-based, and is in characters, not bytes. The original node is retained, its data is just truncated. A new text node is created which contains the remainder of the original data, and is inserted as the next sibling of the original. The new text node is returned.

Syntax

```
xmltextnode* XmlDomSplitText(
   xmlctx *xctx,
   xmltextnode *textnode,
   ub4 offset)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`textnode`	`IN`	`Text` node
`offset`	`IN`	0-based character count at which to split text

Returns

`(xmltextnode *)` new text node

See Also: XmlDomGetCharData, XmlDomAppendData, XmlDomInsertData, XmlDomDeleteData, XmlDomReplaceData

5

Package Range APIs for C

Package Range contains APIs for two interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

DocumentRange Interface

Table 5–1 summarizes the methods of available through the `DocumentRange` interface.

Table 5–1 Summary of DocumentRange Methods; Package Range

Function	Summary
XmlDomCreateRange on page 5-2	Create `Range` object.

XmlDomCreateRange

The only one method of `DocumentRange` interface, used to create a `Range` object.

Syntax

```
xmlrange* XmlDomCreateRange(
   xmlctx *xctx,
   xmlrange *range,
   xmldocnode *doc);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`range`	`IN`	existing NodeIterator, or `NULL` to allocate new
`doc`	`IN`	document to which the new `Range` is attached

Returns

`(xmlrange *)` original or new `Range` object.

Range Interface

Table 5–2 summarizes the methods of available through the `Range` interface.

Table 5–2 Summary of Range Methods; Package Range

Function	Summary
XmlDomRangeClone on page 5-4	Clone a range.
XmlDomRangeCloneContents on page 5-4	Clone contents selected by a range.
XmlDomRangeCollapse on page 5-5	Collapse range to either start point or end point.
XmlDomRangeCompareBoundaryPoints on page 5-5	Compare boundary points of two ranges.
XmlDomRangeDeleteContents on page 5-6	Delete content selected by a range.
XmlDomRangeDetach on page 5-7	Detach a range.
XmlDomRangeExtractContents on page 5-7	Extract contents selected by a range.
XmlDomRangeGetCollapsed on page 5-8	Return whether the range is collapsed.
XmlDomRangeGetCommonAncestor on page 5-8	Return deepest common ancestor node of two boundary points.
XmlDomRangeGetDetached on page 5-9	Return whether the range is detached.
XmlDomRangeGetEndContainer on page 5-9	Return range end container node.
XmlDomRangeGetEndOffset on page 5-10	Return range end offset.
XmlDomRangeGetStartContainer on page 5-10	Return range start container node.
XmlDomRangeGetStartOffset on page 5-11	Return range start offset.
XmlDomRangeIsConsistent on page 5-12	Return whether the range is consistent.
XmlDomRangeSelectNode on page 5-12	Select a node as a range.
XmlDomRangeSelectNodeContents on page 5-13	Define range to select node contents.
XmlDomRangeSetEnd on page 5-13	Set the end point.
XmlDomRangeSetEndBefore on page 5-14	Set the end point before a node.
XmlDomRangeSetStart on page 5-15	Set the start point.
XmlDomRangeSetStartAfter on page 5-15	Set the start point after a node.

Table 5–2 (Cont.) Summary of Range Methods; Package Range

Function	Summary
XmlDomRangeSetStartBefore on page 5-16	Set the start point before a node.

XmlDomRangeClone

Clone a `Range`. Clones the range without affecting the content selected by the original range. Returns `NULL` if an error.

Syntax

```
xmlrange* XmlDomRangeClone(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`range`	`IN`	range object
`xerr`	`OUT`	numeric return code

Returns

`(xmlrange *)` new range that clones the old one

XmlDomRangeCloneContents

Clone contents selected by a range. Clones but does not delete contents selected by a range. Performs the range consistency check and sets `retval` to an error code if an error.

Syntax

```
xmlnode* XmlDomRangeCloneContents(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(xmlnode *) cloned contents

XmlDomRangeCollapse

Collapses the range to either start point or end point. The point where it is collapsed to is assumed to be a valid point in the document which this range is attached to.

Syntax

```
xmlerr XmlDomRangeCollapse(
   xmlctx *xctx,
   xmlrange *range,
   boolean tostart);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
tostart	IN	indicates whether to collapse to start (TRUE) or to end (FALSE)

Returns

(xmlerr) numeric return code

XmlDomRangeCompareBoundaryPoints

Compares two boundary points of two different ranges. Returns -1,0,1 depending on whether the corresponding boundary point of the range (range) is before, equal, or after the corresponding boundary point of the second range (srange). It returns ~(int)0 if two ranges are attached to two different documents or if one of them is detached.

Syntax

```
sb4 XmlDomRangeCompareBoundaryPoints(
   xmlctx *xctx,
   xmlrange *range,
   xmlcmphow how,
   xmlrange *srange,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
how	IN	xmlcmphow value; how to compare
srange	IN	range object with which to compare
xerr	OUT	numeric return code

Returns

(sb4) strcmp-like comparison result

XmlDomRangeDeleteContents

Delete content selected by a range. Deletes content selected by a range. Performs the range consistency check and sets retval to an error code if an error.

Syntax

```
xmlerr XmlDomRangeDeleteContents(
   xmlctx *xctx,
   xmlrange *range);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object

Returns

(xmlerr) numeric return code

XmlDomRangeDetach

Detaches the range from the document and places it (range) in invalid state.

Syntax

```
xmlerr XmlDomRangeDetach(
   xmlctx *xctx,
   xmlrange *range);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object

Returns

(xmlerr) numeric return code

XmlDomRangeExtractContents

Extract contents selected by a range. Clones and deletes contents selected by a range. Performs the range consistency check and sets retval to an error code if an error.

Syntax

```
xmlnode* XmlDomRangeExtractContents(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(xmlnode *) extracted

XmlDomRangeGetCollapsed

Returns TRUE if the range is collapsed and is not detached, otherwise returns FALSE.

Syntax

```
boolean XmlDomRangeGetCollapsed(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(boolean) TRUE if the range is collapsed, FALSE otherwise

XmlDomRangeGetCommonAncestor

Returns deepest common ancestor node of two boundary points of the range if the range is not detached, otherwise returns NULL. It is assumed that the range is in a consistent state.

Syntax

```
xmlnode* XmlDomRangeGetCommonAncestor(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(xmlnode *) deepest common ancestor node [or NULL]

XmlDomRangeGetDetached

Return whether the range is detached. Returns TRUE if the range is detached and is not NULL. Otherwise returns FALSE.

Syntax

```
ub1 XmlDomRangeGetDetached(
   xmlctx *xctx,
   xmlrange *range);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object

Returns

(ub1) TRUE if the range is detached, FALSE otherwise

XmlDomRangeGetEndContainer

Returns range end container node if the range is not detached, otherwise returns NULL.

Syntax

```
xmlnode* XmlDomRangeGetEndContainer(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(xmlnode *) range end container node [or NULL]

XmlDomRangeGetEndOffset

Returns range end offset if the range is not detached, otherwise returns ~(ub4)0 [the maximum ub4 value].

Syntax

```
ub4 XmlDomRangeGetEndOffset(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(ub4) range end offset [or ub4 maximum]

XmlDomRangeGetStartContainer

Returns range start container node if the range is valid and is not detached, otherwise returns NULL.

Syntax

```
xmlnode* XmlDomRangeGetStartContainer(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(xmlnode *) range start container node

XmlDomRangeGetStartOffset

Returns range start offset if the range is not detached, otherwise returns ~(ub4)0 [the maximum ub4 value].

Syntax

```
ub4 XmlDomRangeGetStartOffset(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(ub4) range start offset [or ub4 maximum]

XmlDomRangeIsConsistent

Return whether the range is consistent. Returns TRUE if the range is consistent: both points are under the same root and the start point is before or equal to the end point. Otherwise returns FALSE.

Syntax

```
boolean XmlDomRangeIsConsistent(
   xmlctx *xctx,
   xmlrange *range,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
xerr	OUT	numeric return code

Returns

(ub1) TRUE if the range is consistent, FALSE otherwise

XmlDomRangeSelectNode

Sets the range end point and start point so that the parent node of this node becomes the container node, and the offset is the offset of this node among the children of its parent. The range becomes collapsed. It is assumed that the node is a valid node of its document. If the range is detached, it is ignored, and the range becomes attached.

Syntax

```
xmlerr XmlDomRangeSelectNode(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
node	IN	XML node

Returns

(xmlerr) numeric return code

XmlDomRangeSelectNodeContents

Sets the range start point to the start of the node contents and the end point to the end of the node contents. It is assumed that the node is a valid document node. If the range is detached, it is ignored, and the range becomes attached.

Syntax

```
xmlerr XmlDomRangeSelectNodeContents(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
node	IN	XML node

Returns

(xmlerr) numeric return code

XmlDomRangeSetEnd

Sets the range end point. If it has a root container other than the current one for the range, the range is collapsed to the new position. If the end is set to be at a position before the start, the range is collapsed to that position. Returns xmlerr value.

according to the description where this type is defined. It is assumed that the start point of the range is a valid start point.

Syntax

```
xmlerr XmlDomRangeSetEnd(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node,
   ub4 offset);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
node	IN	XML node
offset	IN	ending offset

Returns

(xmlerr) numeric return code

XmlDomRangeSetEndBefore

Sets the range end point before a node. If it has a root container other than the current one for the range, the range is collapsed to the new position. If the before node sets the end to be at a position before the start, the range is collapsed to new position. Returns xmlerr value according to the description where this type is defined. It is assumed that the start point of the range is a valid start point.

Syntax

```
xmlerr XmlDomRangeSetEndBefore(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node);
```

Parameter	In/Out	Description
xctx	IN	XML context

Parameter	In/Out	Description
range	IN	range object
node	IN	XML node

Returns

(xmlerr) numeric return code

XmlDomRangeSetStart

Sets the range start point. If it has a root container other than the current one for the range, the range is collapsed to the new position. If the start is set to be at a position after the end, the range is collapsed to that position. Returns xmlerr value according to the description where this type is defined. It is assumed that the end point of the range is a valid end point.

Syntax

```
xmlerr XmlDomRangeSetStart(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node,
   ub4 offset);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
node	IN	XML node
offset	IN	starting offset

Returns

(xmlerr) numeric return code

XmlDomRangeSetStartAfter

Sets the range start point after a node. If it has a root container other than the current one for the range, the range is collapsed to the new position. If the after

node sets the start to be at a position after the end, the range is collapsed to new position. Returns xmlerr value according to the description where this type is defined. It is assumed that the end point of the range is a valid end point.

Syntax

```
xmlerr XmlDomRangeSetStartAfter(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object
node	IN	XML node

Returns

(xmlerr) numeric return code

XmlDomRangeSetStartBefore

Sets the range start point before a node. If it has a root container other than the current one for the range, the range is collapsed to the new position with offset 0. If the before node sets the start to be at a position after the end, the range is collapsed to new position. Returns xmlerr value according to the description where this type is defined. It is assumed that the end point of the range is a valid end point.

Syntax

```
xmlerr XmlDomRangeSetStartBefore(
   xmlctx *xctx,
   xmlrange *range,
   xmlnode *node);
```

Parameter	In/Out	Description
xctx	IN	XML context
range	IN	range object

Parameter	In/Out	Description
node	IN	XML node

Returns

(xmlerr) numeric return code

6

Package SAX APIs for C

SAX is a standard interface for event-based XML parsing, developed collaboratively by the members of the XML-DEV mailing list. To use SAX, an `xmlsaxcb` structure is initialized with function pointers and passed to one of the `XmlLoadSax` calls. A pointer to a user-defined context structure is also provided, and will be passed to each SAX function.

This chapter contains the following section:

- SAX Interface

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

SAX Interface

Table 6–1 summarizes the methods of available through the SAX interface.

Table 6–1 Summary of SAX Methods

Function	Summary
XmlSaxAttributeDecl on page 6-3	Receives SAX notification of an attribute's declaration. Oracle
XmlSaxCDATA on page 6-3	Receives SAX notification of CDATA. Oracle extension.
XmlSaxCharacters on page 6-4	Receives SAX notification of character data
XmlSaxComment on page 6-5	Receives SAX notification of a comment.
XmlSaxElementDecl on page 6-5	Receives SAX notification of an element's declaration. Oracle extension.
XmlSaxEndDocument on page 6-6	Receives SAX end-of-document notification.
XmlSaxEndElement on page 6-6	Receives SAX end-of-element notification.
XmlSaxNotationDecl on page 6-7	Receives SAX notification of a notation declaration.
XmlSaxPI on page 6-8	Receives SAX notification of a processing instruction.
XmlSaxParsedEntityDecl on page 6-8	Receives SAX notification of a parsed entity declaration. Oracle extension.
XmlSaxStartDocument on page 6-9	Receives SAX start-of-document notification.
XmlSaxStartElement on page 6-10	Receives SAX start-of-element notification.
XmlSaxStartElementNS on page 6-10	Receives SAX namespace-aware start-of-element notification.
XmlSaxUnparsedEntityDecl on page 6-11	Receives SAX notification of an unparsed entity declaration.
XmlSaxWhitespace on page 6-12	Receives SAX notification of ignorable (whitespace) data.
XmlSaxXmlDecl on page 6-13	Receives SAX notification of an XML declaration. Oracle extension.

XmlSaxAttributeDecl

This event marks an element declaration in the DTD. The element's name and content will be in the data encoding. Note that an attribute may be declared before the element it belongs to!

Syntax

```
xmlerr XmlSaxAttributeDecl(
   void *ctx,
   oratext *elem,
   oratext *attr,
   oratext *body)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
elem	IN	element for which the attribute is declared; data encoding
attr	IN	attribute's name; data encoding
body	IN	body of an attribute declaration

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxAttributeDecl

XmlSaxCDATA

This event handles CDATA, as distinct from Text. If no XmlSaxCDATA callback is provided, the Text callback will be invoked. The data will be in the data encoding, and the returned length is in characters, not bytes. See also XmlSaxWhitespace, which receiving notification about ignorable (whitespace formatting) character data.

Syntax

```
xmlerr XmlSaxCDATA(
   void *ctx,
   oratext *ch,
   size_t len)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
ch	IN	pointer to CDATA; data encoding
len	IN	length of CDATA, in characters

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxWhitespace

XmlSaxCharacters

This event marks character data, either Text or CDATA. If an XmlSaxCDATA callback is provided, then CDATA will be send to that instead; with no XmlSaxCDATA callback, both Text and CDATA go to the XmlSaxCharacters callback. The data will be in the data encoding, and the returned length is in characters, not bytes. See also XmlSaxWhitespace, which receiving notification about ignorable (whitespace formatting) character data.

Syntax

```
xmlerr XmlSaxCharacters(
   void *ctx,
   oratext *ch,
   size_t len)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
ch	IN	pointer to data; data encoding
len	IN	length of data, in characters

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxWhitespace

XmlSaxComment

This event marks a comment in the XML document. The comment's data will be in the data encoding. Oracle extension, not in SAX standard.

Syntax

```
xmlerr XmlSaxComment(
   void *ctx,
   oratext *data)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
data	IN	comment's data; data encoding

Returns

(xmlerr) error code, XMLERR_OK [0] for success

XmlSaxElementDecl

This event marks an element declaration in the DTD. The element's name and content will be in the data encoding.

Syntax

```
xmlerr XmlSaxElementDecl(
   void *ctx,
   oratext *name,
   oratext *content)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
name	IN	element's name
content	IN	element's context model

Returns

`(xmlerr)` error code, `XMLERR_OK [0]` for success

See Also: XmlSaxAttributeDecl

XmlSaxEndDocument

The last SAX event, called once for each document, indicating the end of the document. Matching event is `XmlSaxStartDocument`.

Syntax

```
xmlerr XmlSaxEndDocument(
   void *ctx)
```

Parameter	In/Out	Description
`ctx`	`IN`	user's SAX context

Returns

`(xmlerr)` error code, `XMLERR_OK [0]` for success

See Also: XmlSaxStartDocument

XmlSaxEndElement

This event marks the close of an element; it matches the `XmlSaxStartElement` or `XmlSaxStartElementNS` events. The name is the `tagName` of the element (which may be a qualified name for namespace-aware elements) and is in the data encoding.

Syntax

```
xmlerr XmlSaxEndElement(
   void *ctx,
   oratext *name)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
name	IN	name of ending element; data encoding

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxEndElement

XmlSaxNotationDecl

The even marks the declaration of a notation in the DTD. The notation's name, public ID, and system ID will all be in the data encoding. Both IDs are optional and may be NULL.

Syntax

```
xmlerr XmlSaxNotationDecl(
   void *ctx,
   oratext *name,
   oratext *pubId,
   oratext *sysId)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
name	IN	notation's name; data encoding
pubId	IN	notation's public ID as data encoding, or NULL
sysId	IN	notation's system ID as data encoding, or NULL

Returns

(xmlerr) error code, XMLERR_OK [0] for success

XmlSaxPI

This event marks a `ProcessingInstruction`. The `ProcessingInstruction`s target and data will be in the data encoding. There is always a target, but the data may be `NULL`.

Syntax

```
xmlerr XmlSaxPI(
   void *ctx,
   oratext *target,
   oratext *data)
```

Parameter	In/Out	Description
`ctx`	`IN`	user's SAX context
`target`	`IN`	PI's target; data encoding
`data`	`IN`	PI's data as data encoding, or `NULL`

Returns

`(xmlerr)` error code, `XMLERR_OK [0]` for success

XmlSaxParsedEntityDecl

Marks an parsed entity declaration in the DTD. The parsed entity's name, public ID, system ID, and notation name will all be in the data encoding.

Syntax

```
xmlerr XmlSaxParsedEntityDecl(
   void *ctx,
   oratext *name,
   oratext *value,
   oratext *pubId,
   oratext *sysId,
   boolean general)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
name	IN	entity's name; data encoding
value	IN	entity's value; data encoding
pubId	IN	entity's public ID as data encoding, or NULL
sysId	IN	entity's system ID; data encoding
general	IN	TRUE if general entity, FALSE if parameter entity

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxUnparsedEntityDecl

XmlSaxStartDocument

The first SAX event, called once for each document, indicating the start of the document. Matching event is XmlSaxEndDocument.

Syntax

```
xmlerr XmlSaxStartDocument(
   void *ctx)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxEndDocument

XmlSaxStartElement

This event marks the start of an element. Note this is the original SAX 1 non-namespace-aware version; `XmlSaxStartElementNS` is the SAX 2 namespace-aware version. If both are registered, only the NS version will be called. The element's name will be in the data encoding, as are all the attribute parts. See the functions in the `NamedNodeMap` interface for operating on the attributes map. The matching function is `XmlSaxEndElement` (there is no namespace aware version of this function).

Syntax

```
xmlerr XmlSaxStartElement(
   void *ctx,
   oratext *name,
   xmlnodelist *attrs)
```

Parameter	In/Out	Description
`ctx`	`IN`	user's SAX context
`name`	`IN`	element's name; data encoding
`attrs`	`IN`	`NamedNodeMap` of element's attributes

Returns

`(xmlerr)` error code, `XMLERR_OK [0]` for success

> **See Also:** XmlSaxEndElement, XmlDomGetNodeMapLength in Chapter 4, "Package DOM APIs for C", and XmlDomGetNamedItem in Chapter 4, "Package DOM APIs for C"

XmlSaxStartElementNS

This event marks the start of an element. Note this is the new SAX 2 namespace-aware version; XmlSaxStartElement is the SAX 1 non-namespace-aware version. If both are registered, only the NS version will be called. The element's qualified name, local name, and namespace URI will be in the data encoding, as are all the attribute parts. See the functions in the `NamedNodeMap` interface for

operating on the attributes map. The matching function is `XmlSaxEndElement` (there is no namespace aware version of this function).

Syntax

```
xmlerr XmlSaxStartElementNS(
   void *ctx,
   oratext *qname,
   oratext *local,
   oratext *nsp,
   xmlnodelist *attrs)
```

Parameter	In/Out	Description
`ctx`	`IN`	user's SAX context
`qname`	`IN`	element's qualified name; data encoding
`local`	`IN`	element's namespace local name; data encoding
`nsp`	`IN`	element's namespace URI; data encoding
`attrs`	`IN`	`NodeList` of element's attributes, or `NULL`

Returns

`(xmlerr)` error code, `XMLERR_OK [0]` for success

> **See Also:** XmlSaxStartElement, XmlSaxEndElement, XmlDomGetNodeMapLength in Chapter 4, "Package DOM APIs for C", and XmlDomGetNamedItem in Chapter 4, "Package DOM APIs for C"

XmlSaxUnparsedEntityDecl

Marks an unparsed entity declaration in the DTD, see `XmlSaxParsedEntityDecl` for the parsed entity version. The unparsed entity's name, public ID, system ID, and notation name will all be in the data encoding.

Syntax

```
xmlerr XmlSaxUnparsedEntityDecl(
   void *ctx,
   oratext *name,
```

```
   oratext *pubId,
   oratext *sysId,
   oratext *note)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
name	IN	entity's name; data encoding
pubId	IN	entity's public ID as data encoding, or NULL
sysId	IN	entity's system ID; data encoding
note	IN	entity's notation name; data encoding

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxParsedEntityDecl

XmlSaxWhitespace

This event marks ignorable whitespace data such as newlines, and indentation between lines. The matching function is XmlSaxCharacters, which receives notification of normal character data. The data is in the data encoding, and the returned length is in characters, not bytes.

Syntax

```
xmlerr XmlSaxWhitespace(
   void *ctx,
   oratext *ch,
   size_t len)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
ch	IN	pointer to data; data encoding
len	IN	length of data, in characters

Returns

(xmlerr) error code, XMLERR_OK [0] for success

See Also: XmlSaxCharacters

XmlSaxXmlDecl

This event marks an XML declaration. The XmlSaxStartDocument event is always first; if this callback is registered and an XMLDecl exists, it will be the second event. The encoding flag says whether an encoding was specified. Since the document's own encoding specification may be overridden (or wrong), and the input will be converted to the data encoding anyway, the actual encoding specified in the document is not provided. For the standalone flag, -1 will be returned if it was not specified, otherwise 0 for FALSE, 1 for TRUE.

Syntax

```
xmlerr XmlSaxXmlDecl(
   void *ctx,
   oratext *version,
   boolean encoding,
   sword standalone)
```

Parameter	In/Out	Description
ctx	IN	user's SAX context
version	IN	version string from XMLDecl; data encoding
encoding	IN	whether encoding was specified
standalone	IN	value of the standalone document; < 0 if not specified

Returns

(xmlerr) error code, XMLERR_OK [0] for success

7

Package Schema APIs for C

This C implementation of the XML schema validator follows the W3C XML Schema specification, rev REC-xmlschema-1-20010502. It implements the required behavior of a schema validator for multiple schema documents to be assembled into a schema. This resulting schema can be used to validate a specific instance document.

This chapter contains the following section:

- Schema Interface

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Schema Interface

Table 7–1 summarizes the methods of available through the `Schema` interface.

Table 7–1 Summary of Schema Methods

Function	Summary
XmlSchemaClean on page 7-2	Clean up loaded schemas in a schema context and recycle the schema context.
XmlSchemaCreate on page 7-3	Create and return a schema context.
XmlSchemaDestroy on page 7-3	Destroy a schema context.
XmlSchemaErrorWhere on page 7-4	Returns the location where an error occurred.
XmlSchemaLoad on page 7-4	Load a schema document.
XmlSchemaLoadedList on page 7-5	Return the size and/or list of loaded schema documents.
XmlSchemaSetErrorHandler on page 7-6	Sets an error message handler and its associated context in a schema context
XmlSchemaSetValidateOptions on page 7-6	Set option(s) to be used in the next validation session.
XmlSchemaTargetNamespace on page 7-7	Return target namespace of a given schema document.
XmlSchemaUnload on page 7-8	Unload a schema document.
XmlSchemaValidate on page 7-8	Validate an element node against a schema.
XmlSchemaVersion on page 7-9	Return the version of this schema implementation.

XmlSchemaClean

Clean up loaded schemas in a schema context and recycle the schema context.

Syntax

```
void XmlSchemaClean(
   xsdctx *sctx);
```

Parameter	In/Out	Description
sctx	IN	schema context to be cleaned

See Also: XmlSchemaCreate, XmlSchemaDestroy

XmlSchemaCreate

Return a schema context to be used in other validator APIs. This needs to be paired with an XmlSchemaDestroy.

Syntax

```
xsdctx *XmlSchemaCreate(
   xmlctx *xctx,
   xmlerr *err,
   list);
```

Parameter	In/Out	Description
xctx	IN	XML context
err	OUT	returned error code
list	IN	NULL-terminated list of variable arguments

Returns

(xsdctx *) schema context

See Also: XmlSchemaDestroy, XmlCreate in Chapter 9, "Package XML APIs for C"

XmlSchemaDestroy

Destroy a schema context and free up all its resources.

Syntax

```
void XmlSchemaDestroy(
   xsdctx *sctx);
```

Parameter	In/Out	Description
sctx	IN	schema context to be freed

See Also: XmlSchemaCreate

XmlSchemaErrorWhere

Returns the location (line#, path) where an error occurred.

Syntax

```
xmlerr XmlSchemaErrorWhere(
   xsdctx *sctx,
   ub4 *line,
   oratext **path);
```

Parameter	In/Out	Description
sctx	IN	schema context
line	IN/OUT	line number where error occurred
path	IN/OUT	URL or filespace where error occurred

Returns

(xmlerr) error code

See Also: XmlSchemaSetErrorHandler

XmlSchemaLoad

Load up a schema document to be used in the next validation session. Schema documents can be incrementally loaded into a schema context as long as every loaded schema document is valid. When the last loaded schema turns out to be invalid, you need to clean up the schema context by calling XmlSchemaClean and reload everything all over again including the last schema with appropriate correction.

Syntax

```
xmlerr XmlSchemaLoad(
   xsdctx *sctx,
   oratext *uri,
   list);
```

Parameter	In/Out	Description
sxctx	IN	schema context
uri	IN	URL of schema document; compiler encoding
list	IN	NULL-terminated list of variable arguments

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlSchemaUnload, XmlSchemaLoadedList

XmlSchemaLoadedList

Return only the size of loaded schema documents if list is NULL. If list is not NULL, a list of URL pointers are returned in the user-provided pointer buffer. Note that its user's responsibility to provide a buffer with big enough size.

Syntax

```
ub4 XmlSchemaLoadedList(
   xsdctx *sctx,
   oratext **list);
```

Parameter	In/Out	Description
sctx	IN	schema context
list	IN	address of pointer buffer

Returns

(ub4) list size

See Also: XmlSchemaLoad, XmlSchemaUnload

XmlSchemaSetErrorHandler

Sets an error message handler and its associated context in a schema context. To retrieve useful location information on errors, the address of the schema context must be provided in the error handler context.

Syntax

```
xmlerr XmlSchemaSetErrorHandler(
   xsdctx *sctx,
   XML_ERRMSG_F(
      (*errhdl),
      ectx,
      msg,
      err),
   void *errctx);
```

Parameter	In/Out	Description
sctx	IN	schema context
errhdl	IN	error message handler
errctx	IN	error handler context

Returns

(xmlerr) error code

See Also: XmlSchemaCreate, XmlSchemaErrorWhere, and XML_ERRMSG_F in Chapter 3, "Package Callback APIs for C"

XmlSchemaSetValidateOptions

Set options to be used in the next validation session. Previously set options will remain effective until they are overwritten or reset.

Syntax

```
xmlerr XmlSchemaSetValidateOptions(
```

```
   xsdctx *sctx,
   list);
```

Parameter	In/Out	Description
sctx	IN	schema context
list	IN	NULL-terminated list of variable argument

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlSchemaValidate

XmlSchemaTargetNamespace

Return target namespace of a given schema document identified by its URI. All currently loaded schema documents can be queried. Currently loaded schema documents include the ones loaded through XmlSchemaLoads and the ones loaded through schemaLocation or noNamespaceSchemaLocation hints.

Syntax

```
oratext *XmlSchemaTargetNamespace(
   xsdctx *sctx,
   oratext *uri);
```

Parameter	In/Out	Description
sctx	IN	XML context
uri	IN	URL of the schema document to be queried

Returns

(oratext *) target namespace string; NULL if given document not

See Also: XmlSchemaLoadedList

XmlSchemaUnload

Unload a schema document from the validator. All previously loaded schema documents will remain loaded until they are unloaded. To unload all loaded schema documents, set URI to be `NULL` (this is equivalent to `XmlSchemaClean`). Note that all children schemas associated with the given schema are also unloaded. In this implementation, it only support the following scenarios:

- load, load, ...
- load, load, load, unload, unload, unload, clean, and then repeat.

It doesn't not support: load, load, unload, load, ...

Syntax

```
xmlerr XmlSchemaUnload(
   xsdctx *sctx,
   oratext *uri,
   list);
```

Parameter	In/Out	Description
`sctx`	`IN`	schema context
`uri`	`IN`	URL of the schema document; compiler encoding
`list`	`IN`	`NULL`-terminated list of variable argument

Returns

`(xmlerr)` numeric error code, `XMLERR_OK [0]` on success

See Also: XmlSchemaLoad, XmlSchemaLoadedList

XmlSchemaValidate

Validates an element node against a schema. Schemas used in current session consists of all schema documents specified through `XmlSchemaLoad` and provided as hint(s) through schemaLocation or `noNamespaceSchemaLocation` in the instance document. After the invocation of this routine, all loaded schema documents remain loaded and can be queried by `XmlSchemaLoadedList`. However, they will remain inactive. In the next validation session, inactive schema

documents can be activated by specifying them through XmlSchemaLoad or providing them as hint(s) through schemaLocation or noNamespaceSchemaLocation in the new instance document. To unload a schema document and all its descendants (documents included or imported in a nested manner), use XmlSchemaUnload.

Syntax

```
xmlerr XmlSchemaValidate(
   xsdctx *sctx,
   xmlctx *xctx,
   xmlelemnode *elem);
```

Parameter	In/Out	Description
sctx	IN	schema context
xctx	IN	XML top-level context
elem	IN	element node in the doc, to be validated

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlSchemaSetValidateOptions

XmlSchemaVersion

Return the version of this schema implementation.

Syntax

```
oratext *XmlSchemaVersion();
```

Returns

(oratext *) version string [compiler encoding]

8

Package Traversal APIs for C

Package Traversal contains APIs for four interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

DocumentTraversal Interface

Table 8–1 summarizes the methods of available through the DocumentTraversal interface.

Table 8–1 Summary of DocumentTraversal Methods; Traversal Package

Function	Summary
XmlDomCreateNodeIter on page 8-2	Create node iterator object.
XmlDomCreateTreeWalker on page 8-3	Create a tree walker object.

XmlDomCreateNodeIter

One of two methods of `DocumentTraversal` interface, used to create a `NodeIterator` object. This method is identical to XmlDomCreateTreeWalker except for the type of object returned.

The `whatToShow` argument is a mask of flag bits, one for each node type. The value `XMLDOM_SHOW_ALL` passes all node types through, otherwise only the types whose bits are set will be passed.

`Entity` reference expansion is controlled by the `entrefExpansion` flag. If `TRUE`, entity references are replaced with their final content; if `FALSE`, entity references are left as nodes.

Syntax

```
xmliter* XmlDomCreateNodeIter(
   xmlctx *xctx,
   xmliter *iter,
   xmlnode *root,
   xmlshowbits whatToShow,
   XMLDOM_ACCEPT_NODE_F(
      (*nodeFilter),
      xctx,
      node),
   boolean entrefExpand);
```

Parameter	In/Out	Description
xctx	IN	XML context
iter	IN	existing NodeIterator to set, NULL to create
xerr	IN	root node for NodeIterator
whatToShow	IN	mask of XMLDOM_SHOW_*XXX* flag bits
nodeFilter	IN	node filter to be used, NULL if none
xerr	IN	whether to expand entity reference nodes

Returns

(xmliter *) original or new NodeIterator object

See Also: XmlDomCreateTreeWalker

XmlDomCreateTreeWalker

One of two methods of DocumentTraversal interface, used to create a TreeWalker object. This method is identical to XmlDomCreateNodeIter except for the type of object returned.

The whatToShow argument is a mask of flag bits, one for each node type. The value XMLDOM_SHOW_ALL passes all node types through, otherwise only the types whose bits are set will be passed.

Entity reference expansion is controlled by the entrefExpansion flag. If TRUE, entity references are replaced with their final content; if FALSE, entity references are left as nodes.

Syntax

```
xmlwalk* XmlDomCreateTreeWalker(
   xmlctx *xctx,
   xmlwalk* walker,
   xmlnode *root,
   xmlshowbits whatToShow,
   XMLDOM_ACCEPT_NODE_F(
      (*nodeFilter),
      xctx,
      node),
   boolean entrefExpansion);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	existing TreeWalker to set, NULL to create
xerr	IN	root node for TreeWalker
whatToShow	IN	mask of XMLDOM_SHOW_*XXX* flag bits
nodeFilter	IN	node filter to be used, NULL if none
xerr	IN	whether to expand entity reference nodes

Returns

(xmlwalk *) new TreeWalker object

See Also: XmlDomCreateNodeIter

NodeFilter Interface

Table 8–2 summarizes the methods of available through the `NodeFilter` interface.

Table 8–2 Summary of NodeFilter Methods; Traversal Package

Function	Summary
XMLDOM_ACCEPT_NODE_F on page 8-5	Perform user-defined filtering action on node.

XMLDOM_ACCEPT_NODE_F

Sole method of `NodeFilter` interface. Given a node and a filter, determines the filtering action to perform.

This function pointer is passed to the node iterator/tree walker methods, as needed.

Values for xmlerr are:

- `XMLERR_OK` Accept the node. Navigation methods defined for `NodeIterator` or `TreeWalker` will return this node.
- `XMLERR_FILTER_REJECT` Reject the node. Navigation methods defined for `NodeIterator` or `TreeWalker` will not return this node. For `TreeWalker`, the children of this node will also be rejected. `NodeIterators` treat this as a synonym for `XMLDOM_FILTER_SKIP`
- `XMLERR_FILTER_SKIP` Skip this single node. Navigation methods defined for `NodeIterator` or `TreeWalker` will not return this node. For both `NodeIterator` and `TreeWalker`, the children of this node will still be considered.

Syntax

```
#define XMLDOM_ACCEPT_NODE_F(func, xctx, node)
xmlerr func(
   xmlctx *xctx,
   xmlnode *node)
```

Parameter	In/Out	Description
xctx	IN	XML context

Parameter	In/Out	Description
node	IN	node to test

Returns

(xmlerr) filtering result

NodeIterator Interface

Table 8–3 summarizes the methods of available through the `NodeIterator` interface.

Table 8–3 Summary of NodeIterator Methods; Package Traversal

Function	Summary
XmlDomIterDetach on page 8-7	Detach a node iterator (deactivate it).
XmlDomIterNextNode on page 8-8	Returns next node for iterator.
XmlDomIterPrevNode on page 8-8	Returns previous node for iterator.

XmlDomIterDetach

Detaches the `NodeIterator` from the set which it iterated over, releasing any resources and placing the iterator in the `INVALID` state. After detach has been invoked, calls to `XmlDomIterNextNode` or `XmlDomIterPrevNode` will raise the exception `XMLERR_ITER_DETACHED`.

Syntax

```
xmlerr XmlDomIterDetach(
   xmlctx *xctx,
   xmliter *iter);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`iter`	`IN`	node iterator object

See Also: XmlDomIterNextNode, XmlDomIterPrevNode

XmlDomIterNextNode

Returns the next node in the set and advances the position of the iterator in the set. After a node iterator is created, the first call to `XmlDomIterNextNode` returns the first node in the set. It assumed that the reference node (current iterator position) is never deleted. Otherwise, changes in the underlying DOM tree do not invalidate the iterator.

Syntax

```
xmlnode* XmlDomIterNextNode(
   xmlctx *xctx,
   xmliter *iter,
   xmlerr *xerr);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`iter`	`IN`	node iterator object
`xerr`	`OUT`	numeric return error code

Returns

`(xmlnode *)` next node in set being iterated over [or `NULL`]

See Also: XmlDomIterPrevNode, XmlDomIterDetach

XmlDomIterPrevNode

Returns the previous node in the set and moves the position of the iterator backward in the set.

Syntax

```
xmlnode* XmlDomIterPrevNode(
   xmlctx *xctx,
   xmliter *iter,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
iter	IN	node iterator object
xerr	OUT	numeric return error code

Returns

(xmlnode *) previous node in set being iterated over [or NULL]

See Also: XmlDomIterNextNode, XmlDomIterDetach

TreeWalker Interface

Table 8–4 summarizes the methods of available through the `TreeWalker` interface.

Table 8–4 Summary of TreeWalker Methods; Traversal Package

Function	Summary
XmlDomWalkerFirstChild on page 8-10	Return first visible child of current node.
XmlDomWalkerGetCurrentNode on page 8-11	Return current node.
XmlDomWalkerGetRoot on page 8-11	Return root node.
XmlDomWalkerLastChild on page 8-12	Return last visible child of current node.
XmlDomWalkerNextNode on page 8-13	Return next visible node.
XmlDomWalkerNextSibling on page 8-13	Return next sibling node.
XmlDomWalkerParentNode on page 8-14	Return parent node.
XmlDomWalkerPrevNode on page 8-15	Return previous node.
XmlDomWalkerPrevSibling on page 8-15	Return previous sibling node.
XmlDomWalkerSetCurrentNode on page 8-16	Set current node.
XmlDomWalkerSetRoot on page 8-17	Set the root node.

XmlDomWalkerFirstChild

Moves the `TreeWalker` to the first visible child of the current node, and returns the new node. If the current node has no visible children, returns `NULL`, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerFirstChild(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) first visible child [or NULL]

See Also: XmlDomWalkerLastChild

XmlDomWalkerGetCurrentNode

Return (get) current node, or NULL on error.

Syntax

```
xmlnode* XmlDomWalkerGetCurrentNode(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) current node

XmlDomWalkerGetRoot

Return (get) root node, or NULL on error. Since the current node can be removed from under the root node together with a subtree where it belongs to, the current root node in a walker might have no relation to the current node any more. The TreeWalker iterations are based on the current node. However, the root node

defines the space of an iteration. This function checks if the root node is still in the root node (ancestor) relation to the current node. If so, it returns this root node. Otherwise, it finds the root of the tree where the current node belongs to, and sets and returns this root as the root node of the walker. It returns NULL if the walker is a NULL pointer.

Syntax

```
xmlnode* XmlDomWalkerGetRoot(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) root node

XmlDomWalkerLastChild

Moves the TreeWalker to the last visible child of the current node, and returns the new node. If the current node has no visible children, returns NULL, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerLastChild(
   xmlctx *xctx,
   xmlwalk *walker,
    xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object

Parameter	In/Out	Description
xerr	OUT	numeric return error code

Returns

(xmlnode *) last visible children [or NULL]

XmlDomWalkerNextNode

Moves the TreeWalker to the next visible node in document order relative to the current node, and returns the new node. If the current node has no next node, or if the search for the next node attempts to step upward from the TreeWalker's root node, returns NULL, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerNextNode(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) next node [or NULL]

See Also: XmlDomWalkerPrevNode, XmlDomWalkerNextSibling, XmlDomWalkerPrevSibling

XmlDomWalkerNextSibling

Moves the TreeWalker to the next sibling of the current node, and returns the new node. If the current node has no visible next sibling, returns NULL, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerNextSibling(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) next sibling [or NULL]

See Also: XmlDomWalkerNextNode, XmlDomWalkerPrevNode, XmlDomWalkerPrevSibling

XmlDomWalkerParentNode

Moves to and returns the closest visible ancestor node of the current node. If the search for the parent node attempts to step upward from the TreeWalker's root node, or if it fails to find a visible ancestor node, this method retains the current position and returns null.

Syntax

```
xmlnode* XmlDomWalkerParentNode(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

`(xmlnode *)` parent node [or `NULL`]

XmlDomWalkerPrevNode

Moves the `TreeWalker` to the previous visible node in document order relative to the current node, and returns the new node. If the current node has no previous node, or if the search for the previous node attempts to step upward from the `TreeWalker`'s root node, returns `NULL`, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerPrevNode(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`walker`	`IN`	`TreeWalker` object
`xerr`	`OUT`	numeric return error code

Returns

`(xmlnode *)` previous node [or `NULL`]

See Also: XmlDomWalkerNextNode, XmlDomWalkerNextSibling, XmlDomWalkerPrevSibling

XmlDomWalkerPrevSibling

Moves the `TreeWalker` to the previous sibling of the current node, and returns the new node. If the current node has no visible previous sibling, returns `NULL`, and retains the current node.

Syntax

```
xmlnode* XmlDomWalkerPrevSibling(
   xmlctx *xctx,
```

```
   xmlwalk *walker,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
xerr	OUT	numeric return error code

Returns

(xmlnode *) previous sibling [or NULL]

See Also: XmlDomWalkerNextNode, XmlDomWalkerPrevNode, XmlDomWalkerNextSibling

XmlDomWalkerSetCurrentNode

Sets and returns new current node. It also checks if the root node is an ancestor of the new current node. If not it does not set the current node, returns NULL, and sets retval to XMLDOM_WALKER_BAD_NEW_CUR. Returns NULL if an error.

Syntax

```
xmlnode* XmlDomWalkerSetCurrentNode(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlnode *node,
   xmlerr *xerr);
```

Parameter	In/Out	Description
xctx	IN	XML context
walker	IN	TreeWalker object
node	IN	new current node
xerr	OUT	numeric return error code

Returns

`(xmlnode *)` new current node

XmlDomWalkerSetRoot

Set the root node. Returns new root node if it is an ancestor of the current node. If not it signals an error and checks if the current root node is an ancestor of the current node. If yes it returns it. Otherwise it sets the root node to and returns the root of the tree where the current node belongs to. It returns `NULL` if the walker or the root node parameter is a `NULL` pointer.

Syntax

```
xmlnode* XmlDomWalkerSetRoot(
   xmlctx *xctx,
   xmlwalk *walker,
   xmlnode *node,
   xmlerr *xerr);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`walker`	`IN`	`TreeWalker` object
`node`	`IN`	new root node
`xerr`	`OUT`	numeric return error code

Returns

`(xmlnode *)` new root node

9

Package XML APIs for C

This C implementation of the XML processor (or parser) follows the W3C XML specification (rev REC-xml-19980210) and implements the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

This chapter contains the following section:

- XML Interface

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XML Interface

Table 9–1 summarizes the methods of available through the XML interface.

Table 9–1 Summary of XML Methods

Function	Summary
XmlAccess on page 9-2	Set access method callbacks for URL.
XmlCreate on page 9-4	Create an XML Developer's Toolkit xmlctx.
XmlCreateDTD on page 9-6	Create DTD.
XmlCreateDocument on page 9-7	Create Document (node).
XmlDestroy on page 9-7	Destroy an xmlctx.
XmlFreeDocument on page 9-8	Free a document (releases all resources).
XmlGetEncoding on page 9-8	Returns data encoding in use by XML context.
XmlHasFeature on page 9-9	Determine if DOM feature is implemented.
XmlIsSimple on page 9-10	Returns single-byte (simple) characterset flag.
XmlIsUnicode on page 9-10	Returns XmlIsUnicode (simple) characterset flag.
XmlLoadDom on page 9-11	Load (parse) an XML document and produce a DOM.
XmlLoadSax on page 9-13	Load (parse) an XML document from and produce SAX events.
XmlLoadSaxVA on page 9-13	Load (parse) an XML document from and produce SAX events [varargs].
XmlSaveDom on page 9-14	Saves (serializes, formats) an XML document.
XmlVersion on page 9-15	Returns version string for XDK.

XmlAccess

Sets the open/read/close callbacks used to load data for a specific URL access method. Overrides the built-in data loading functions for HTTP, FTP, and so on, or provides functions to handle new types, such as UNKNOWN.

Syntax

```
xmlerr XmlAccess(
   xmlctx *xctx,
   xmlurlacc access,
   void *userctx,
   XML_ACCESS_OPEN_F(
      (*openf),
      ctx,
      uri,
      parts,
      length,
      uh),
   XML_ACCESS_READ_F(
      (*readf),
      ctx,
      uh,
      data,
      nraw,
      eoi),
   XML_ACCESS_CLOSE_F(
      (*closef),
      ctx,
      uh));
```

Parameter	In/Out	Description
xctx	IN	XML context
access	IN	URL access method
userctx	IN	user-defined context passed to callbacks
openf	IN	open-access callback function
readf	IN	read-access callback function
closef	IN	close-access callback function

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlLoadDom, XmlLoadSax

XmlCreate

Create an XML Developer's Toolkit `xmlctx`. Properties common to all xmlctx's (both XDK and XMLType) are:

- "`data_encoding`", name of data encoding) The encoding in which XML data will be presented through DOM and SAX. If not specified, the default is UTF-8 (or UTF-E on EBCDIC platforms). Note that single-byte encodings such as EBCDIC or ISO-8859 are substantially faster than multibyte encodings like UTF-8; Unicode (UTF-16) uses more memory but has better performance than multibyte.
- `BEGIN_NO_DOC` ("`data_lid`", data encoding lid) The data encoding specified as an NLS `lx_langid`; the matching NLS global area must also be specified. `END_NO_DOC`
- "`default_input_encoding`", name of default input encoding) If the encoding of an input document cannot be automatically determined through BOM, `XMLDecl`, protocol header, and so on, then this encoding will be assumed.
- `BEGIN_NO_DOC` ("`default_input_lid`", default input encoding lid) The default input encoding specified as an NLS `lx_langid`; the matching NLS global area must also be specified. `END_NO_DOC`
- "`error_language`", error language or language.encoding) The language (and optional encoding) in which error messages are created. The default is American with UTF-8 encoding. To specify just the language, give the name of the language and nothing else ("American"); To also specify the encoding, add a dot and the Oracle name of the encoding ("American.WE8ISO8859P1").
- "`error_handler`", function pointer, see `XML_ERRMSG_F`) Default behavior on errors is to output the formatted message to `stderr`. If an error handler is provided, the formatted message will be passed to it instead of being printed.
- "`error_context`", user-defined context for error handler) This is a context pointer to be passed to the error handler function. It's meaning is user-defined; it is just specified here and passed along when an error occurs.
- "`input_encoding`", name of forced input encoding) The forced input encoding for input documents. Used to override a document's `XMLDecl`, and so on, and always interpret it in the given encoding. **Use of this feature is strongly discouraged.** It should be not necessary in normal use, as BOMs, `XMLDecls`, and so on, when existing, should be correct.

- BEGIN_NO_DOC ("input_lid", INLID, POINTER), The forced input encoding
- ("lpu_context", lpu context) The LPU context used for URL data loading and access-method hooking. If one is not provided, it will be made for you.
- ("lml_context", LMLCTX, POINTER), The LML context used for low-level memory allocation. If not provided, one will be made. From the outside, end-users have to set memory_alloc, memory_free, and so on. END_NO_DOC
- ("memory_alloc", low-level memory allocation function) Low-level memory allocation function, if malloc is not to be used. If provided, the matching free function must also be given. See XML_ALLOC_F.
- ("memory_free", low-level memory freeing function) Low-level memory freeing function, if free is not to be used. Matches the alloc function.
- ("memory_context", user-defined memory context) User-defined memory context which is passed to the alloc and free functions. Its definition and use is entirely up to the user; it is just set here and passed to the callbacks.
- BEGIN_NO_DOC ("nls_global_area", NLS global area, lx_glo) If any encoding are specified as NLS lids, the matching NLS global area must also be specified. END_NO_DOC
- The XDK has properties of its own, that only apply to an XDK type xmlctx (the previous properties were all general and applied to all xmlctx's).
- ("input_buffer_size", size in characters of input buffer) This is the basic I/O buffer size. Default is 256K, minimum is 4K and maximum is 4MB. Depending on the encoding, 1, 2 or 3 of these buffers may be needed. Note size is in characters, not bytes. If the buffer holds Unicode data, it will be twice as large.
- ("memory_block_size", size in bytes of memory allocation unit) This is the size of chunk the high-level memory package will request from the low-level allocator; i.e., the basic unit of memory allocation. Default is 64K, minimum is 16K and maximum is 256K.

Syntax

```
xmlctx *XmlCreate(
   xmlerr *err,
   oratext *name,
   list);
```

Parameter	In/Out	Description
err	OUT	returned error code
access	IN	name of context, for debugging
list	IN	NULL-terminated list of variable arguments

Returns

(xmlctx *) created xmlctx [or NULL on error with err set]

> **See Also:** XmlDestroy, XML_ERRMSG_F in Chapter 3, "Package Callback APIs for C"

XmlCreateDTD

Create DTD.

Syntax

```
xmldocnode* XmlCreateDTD(
   xmlctx *xctx
   oratext *qname,
   oratext *pubid,
   oratext *sysid,
   xmlerr *err)
```

Parameter	In/Out	Description
xctx	IN	XML context
qname	IN	qualified name
pubid	IN	external subset public identifier
sysid	IN	external subset system identifier
err	OUT	returned error code

Returns

(xmldtdnode *) new DTD node

XmlCreateDocument

Creates the initial top-level DOCUMENT node and its supporting infrastructure. If a qualified name is provided, a an element with that name is created and set as the document's root element.

Syntax

```
xmldocnode* XmlCreateDocument(
   xmlctx *xctx,
   oratext *uri,
   oratext *qname,
   xmldtdnode *dtd,
   xmlerr *err)
```

Parameter	In/Out	Description
xctx	IN	XML context
uri	IN	namespace URI of root element to create, or NULL
qname	IN	qualified name of root element, or NULL if none
dtd	IN	associated DTD node
err	OUT	returned error code

Returns

(xmldocnode *) new Document object.

XmlDestroy

Destroys an xmlctx

Syntax

```
void XmlDestroy(
   xmlctx *xctx)
```

Parameter	In/Out	Description
xctx	IN	XML context

See Also: XmlCreate

XmlFreeDocument

Destroys a document created by `XmlCreateDocument` or through one of the Load functions. Releases all resources associated with the document, which is then invalid.

Syntax

```
void XmlFreeDocument(
   xmlctx *xctx,
   xmldocnode *doc)
```

Parameter	In/Out	Description
xctx	IN	XML context
doc	IN	document to free

See Also: XmlCreateDocument, XmlLoadDom

XmlGetEncoding

Returns data encoding in use by XML context. Ordinarily, the data encoding is chosen by the user, so this function is not needed. However, if the data encoding is not specified, and allowed to default, this function can be used to return the name of that default encoding.

Syntax

```
oratext *XmlGetEncoding(
   xmlctx *xctx);
```

Parameter	In/Out	Description
xctx	IN	XML context

Returns

`(oratext *)` name of data encoding

> **See Also:** XmlDomGetDecl in Chapter 4, "Package DOM APIs for C", XmlIsSimple, XmlIsUnicode

XmlHasFeature

Determine if a DOM feature is implemented. Returns `TRUE` if the feature is implemented in the specified version, `FALSE` otherwise.

In level 1, the legal values for package are 'HTML' and 'XML' (case-insensitive), and the version is the string "1.0". If the version is not specified, supporting any version of the feature will cause the method to return `TRUE`.

- DOM 1.0 features are "XML" and "HTML".
- DOM 2.0 features are "Core", "XML", "HTML", "Views", "StyleSheets", "CSS", "CSS2", "Events", "UIEvents", "MouseEvents", "MutationEvents", "HTMLEvents", "Range", "Traversal"

Syntax

```
boolean XmlHasFeature(
   xmlctx *xctx,
   oratext *feature,
   oratext *version)
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`feature`	`IN`	package name of the feature to test
`version`	`IN`	version number of the package name to test

Returns

`(boolean)` feature is implemented?

XmlIsSimple

Returns a flag saying whether the context's data encoding is "simple", single-byte for each character, like ASCII or EBCDIC.

Syntax

```
boolean XmlIsSimple(
   xmlctx *xctx);
```

Parameter	In/Out	Description
xctx	IN	XML context

Returns

(boolean) TRUE of data encoding is "simple", FALSE otherwise

See Also: XmlGetEncoding, XmlIsUnicode

XmlIsUnicode

Returns a flag saying whether the context's data encoding is Unicode, UTF-16, with two-byte for each character.

Syntax

```
boolean XmlIsUnicode(
   xmlctx *xctx);
```

Parameter	In/Out	Description
xctx	IN	XML context

Returns

(boolean) TRUE of data encoding is Unicode, FALSE otherwise

See Also: XmlGetEncoding, XmlIsSimple

XmlLoadDom

Loads (parses) an XML document from an input source and creates a DOM. The root document node is returned on success, or NULL on failure (with err set).

The function takes two fixed arguments, the xmlctx and an error return code, then zero or more (property, value) pairs, then NULL.

SOURCE Input source is set by one of the following mutually exclusive properties (choose one):

- ("uri", document URI) [compiler encoding]
- ("file", document filesystem path) [compiler encoding]
- ("buffer", address of buffer, "buffer_length", # bytes in buffer)
- ("stream", address of stream object, "stream_context", pointer to stream object's context)
- ("stdio", FILE* stream)

PROPERTIES Additional properties:

- ("dtd", DTD node) DTD for document
- ("base_uri", document base URI) for documents loaded from other sources than a URI, sets the effective base URI. the document's base URI is needed in order to resolve relative URI include, import, and so on.
- ("input_encoding", encoding name) forced input encoding [name]
- ("default_input_encoding", encoding_name) default input encoding to assume if document is not self-describing (no BOM, protocol header, XMLDecl, and so on)
- ("schema_location", string) schemaLocation of schema for this document. used to figure optimal layout when loading documents into a database
- ("validate", boolean) when TRUE, turns on DTD validation; by default, only well-formedness is checked. note that schema validation is a separate beast.
- ("discard_whitespace", boolean) when TRUE, formatting whitespace between elements (newlines and indentation) in input documents is discarded. by default, ALL input characters are preserved.

- ("dtd_only", boolean) when TRUE, parses an external DTD, not a complete XML document.
- ("stop_on_warning", boolean) when TRUE, warnings are treated the same as errors and cause parsing, validation, and so on, to stop immediately. by default, warnings are issued but the game continues.
- ("warn_duplicate_entity", boolean) when TRUE, entities which are declared more than once will cause warnings to be issued. the default is to accept the first declaration and silently ignore the rest.
- ("no_expand_char_ref", boolean) when TRUE, causes character references to be left unexpanded in the DOM data. ordinarily, character references are replaced by the character they represent. however, when a document is saved those characters entities do not reappear. to way to ensure they remain through load and save is to not expand them.
- ("no_check_chars", boolean) when TRUE, omits the test of XML [2] Char production: all input characters will be accepted as valid

Syntax

```
xmldocnode *XmlLoadDom(
   xmlctx *xctx,
   xmlerr *err,
   list);
```

Parameter	In/Out	Description
xctx	IN	XML context
err	OUT	returned error code
list	IN	NULL-terminated list of variable arguments

Returns

(xmldocnode *) document node on success [NULL on failure with err set]

See Also: XmlSaveDom

XmlLoadSax

Loads (parses) an XML document from an input source and generates a set of SAX events (as user callbacks). Input sources and basic set of properties is the same as for XmlLoadDom.

Syntax

```
xmlerr XmlLoadSax(
   xmlctx *xctx,
   xmlsaxcb *saxcb,
   void *saxctx,
   list);
```

Parameter	In/Out	Description
xctx	IN	XML context
saxcb	IN	SAX callback structure
saxctx	IN	context passed to SAX callbacks
list	IN	NULL-terminated list of variable arguments

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

XmlLoadSaxVA

Loads (parses) an XML document from an input source and generates a set of SAX events (as user callbacks). Input sources and basic set of properties is the same as for XmlLoadDom.

Syntax

```
xmlerr XmlLoadSaxVA(
   xmlctx *xctx,
   xmlsaxcb *saxcb,
   void *saxctx,
   va_list va);
```

Parameter	In/Out	Description
xctx	IN	XML context
saxcb	IN	SAX callback structure
saxctx	IN	context passed to SAX callbacks
va	IN	NULL-terminated list of variable arguments

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

XmlSaveDom

Serializes document or subtree to the given destination and returns the number of bytes written; if no destination is provided, just returns formatted size but does not output.

If an output encoding is specified, the document will be re-encoded on output; otherwise, it will be in its existing encoding.

The top level is indented step*level spaces, the next level step*(level+1) spaces, and so on.

When saving to a buffer, if the buffer overflows, 0 is returned and err is set to XMLERR_SAVE_OVERFLOW.

DESTINATION Output destination is set by one of the following mutually exclusive properties (choose one):

- ("uri", document URI) POST, PUT? [compiler encoding]
- ("file", document filesystem path) [compiler encoding]
- ("buffer", address of buffer, "buffer_length", # bytes in buffer)
- ("stream", address of stream object, "stream_context", pointer to stream object's context)

PROPERTIES Additional properties:

- ("output_encoding", encoding name) name of final encoding for document. unless specified, saved document will be in same encoding as xmlctx.
- ("indent_step", unsigned) spaces to indent each level of output. default is 4, 0 means no indentation.

- ("indent_level", unsigned) initial indentation level. default is 0, which means no indentation, flush left.
- ("xmldecl", boolean) include an XMLDecl in the output document. ordinarily an XMLDecl is output for a compete document (root node is DOC).
- ("bom", boolean) input a BOM in the output document. usually the BOM is only needed for certain encodings (UTF-16), and optional for others (UTF-8). causes optional BOMs to be output.
- ("prune", boolean) prunes the output like the unix 'find' command; does not not descend to children, just prints the one node given.

Syntax

```
ubig_ora XmlSaveDom(
   xmlctx *xctx,
   xmlerr *err,
   xmlnode *root,
   list);
```

Parameter	In/Out	Description
xctx	IN	XML context
err	OUT	error code on failure
root	IN	root node or subtree to save
list	IN	NULL-terminated list of variable arguments

Returns

(ubig_ora) number of bytes written to destination

See Also: XmlLoadDom

XmlVersion

Returns the version string for the XDK

Syntax

```
oratext *XmlVersion();
```

Returns

`(oratext *)` version string

10

Package XPath APIs for C

XPath methods process XPath related types and interfaces.

This chapter contains this section:

- XPath Interface

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XPath Interface

Table 10–1 summarizes the methods of available through the `XPath` interface.

Table 10–1 Summary of XPath Methods

Function	Summary
XmlXPathCreateCtx on page 10-2	Create an XPath context.
XmlXPathDestroyCtx on page 10-3	Destroy an XPath context.
XmlXPathEval on page 10-3	Evaluate XPath expression.
XmlXPathGetObjectBoolean on page 10-4	Get boolean value of XPath object.
XmlXPathGetObjectFragment on page 10-4	Get fragment value of XPath object.
XmlXPathGetObjectNSetNode on page 10-5	Get node from nodeset type XPath object.
XmlXPathGetObjectNSetNum on page 10-5	Get number of nodes in nodeset type XPath object.
XmlXPathGetObjectNumber on page 10-6	Get number from XPath object.
XmlXPathGetObjectString on page 10-7	Get string from XPath object.
XmlXPathGetObjectType on page 10-7	Get XPath object type.
XmlXPathParse on page 10-8	Parse XPath expression.

XmlXPathCreateCtx

Create an `XPath` context

Syntax

```
xpctx* XmlXPathCreateCtx(
   xmlctx *xsl,
   oratext *baseuri,
   xmlnode *ctxnode,
   ub4 ctxpos,
   ub4 ctxsize);
```

Parameter	In/Out	Description
xsl	IN	XSL stylesheet as xmldoc object
baseuri	IN	base URI used by document, if any
ctxnode	IN	current context position
ctxpos	IN	current context size
ctxsize	IN	current context node

Returns

(xpctx *) XPath context or NULL on error

XmlXPathDestroyCtx

Destroy an XPath context.

Syntax

```
void XmlXPathDestroyCtx(
   xpctx *xslxpctx);
```

Parameter	In/Out	Description
xslxpctx	IN	XPath context object

XmlXPathEval

Evaluate XPath expression.

Syntax

```
xpobj *XmlXPathEval(
   xpctx *xctx,
   xpexpr *exprtree,
   xmlerr *err);
```

Parameter	In/Out	Description
xctx	IN	XPath context
exprtree	IN	parsed XPath expression tree
err	OUT	error code

Returns

(xpobj *) result XPath object or NULL on error

XmlXPathGetObjectBoolean

Get boolean value of XPath object

Syntax

```
boolean XmlXPathGetObjectBoolean(
   xpobj *obj);
```

Parameter	In/Out	Description
obj	IN	XPath object

Returns

(boolean) truth value

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNum, XmlXPathGetObjectNSetNode, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathGetObjectFragment

Get boolean value of XPath object

Syntax

```
xmlnode* XmlXPathGetObjectFragment(
   xpobj *obj);
```

Parameter	In/Out	Description
obj	IN	XPath object

Returns

(boolean) truth value

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNum, XmlXPathGetObjectNSetNode, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathGetObjectNSetNode

Get node from nodeset-type XPath object

Syntax

```
xmlnode *XmlXPathGetObjectNSetNode(
   xpobj *obj,
   ub4 i);
```

Parameter	In/Out	Description
obj	IN	XPath object
i	IN	node index in nodeset

Returns

(xmlnode *) The object type or values.

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNum, XmlXPathGetObjectString, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathGetObjectNSetNum

Get number of nodes in nodeset-type XPath object

Syntax

```
ub4 XmlXPathGetObjectNSetNum(
   xpobj *obj);
```

Parameter	In/Out	Description
obj	IN	XPath object

Returns

(ub4) number of nodes in nodeset

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNode, XmlXPathGetObjectString, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathGetObjectNumber

Get number from XPath object

Syntax

```
double XmlXPathGetObjectNumber(
   xpobj *obj);
```

Parameter	In/Out	Description
obj	IN	XPath object

Returns

(double) number

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNum, XmlXPathGetObjectNSetNode, XmlXPathGetObjectString, XmlXPathGetObjectBoolean

XmlXPathGetObjectString

Get string from `XPath` object

Syntax

```
oratext *XmlXPathGetObjectString(
   xpobj *obj);
```

Parameter	In/Out	Description
`obj`	`IN`	XPath object

Returns

`(oratext *)` string

See Also: XmlXPathGetObjectType, XmlXPathGetObjectNSetNum, XmlXPathGetObjectNSetNode, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathGetObjectType

Get `XPath` object type

Syntax

```
xmlxslobjtype XmlXPathGetObjectType(
   xpobj *obj);
```

Parameter	In/Out	Description
`obj`	`IN`	XPath object

Returns

`(xmlxslobjtype)` type-code for object

See Also: XmlXPathGetObjectNSetNum, XmlXPathGetObjectNSetNode, XmlXPathGetObjectString, XmlXPathGetObjectNumber, XmlXPathGetObjectBoolean

XmlXPathParse

Parse `XPath` expression.

Syntax

```
xpexpr* XmlXPathParse(
   xpctx *xctx,
   oratext *expr,
   xmlerr * err);
```

Parameter	In/Out	Description
`xctx`	`IN`	XPath context object
`expr`	`IN`	XPath expression
`err`	`OUT`	error code

Returns

`(xpexpr *) XPath` expression parse tree or `NULL` on error

11

Package XPointer APIs for C

Package `XPointer` contains APIs for three interfaces.

This chapter contains these sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XPointer Interface

Table 11–1 summarizes the methods of available through the `XPointer` interface.

Table 11–1 Summary of XPointer Methods; Package XPointer

Function	Summary
XmlXPointerEval on page 11-2	Evaluates xpointer string.

XmlXPointerEval

Parses and evaluates xpointer string and calculates locations in the document.

Syntax

```
xmlxptrlocset* XmlXPointerEval(
   xmldocnode* doc,
   oratext* xptrstr);
```

Parameter	In/Out	Description
`doc`	`IN`	document node of the corresponding DOM tree
`xptrstr`	`IN`	xpointer string

Returns

`(xmlxptrlocset *)` calculated location set

XPtrLoc Interface

Table 11–2 summarizes the methods of available through the `XPtrLoc` interface.

Table 11–2 Summary of XPtrLoc Methods; Package XPointer

Function	Summary
XmlXPtrLocGetNode on page 11-3	Returns Xml node from `XPtrLoc`.
XmlXPtrLocGetPoint on page 11-3	Returns Xml point from `XPtrLoc`.
XmlXPtrLocGetRange on page 11-4	Returns Xml range from `XPtrLoc`.
XmlXPtrLocGetType on page 11-4	Returns type of `XPtrLoc`.
XmlXPtrLocToString on page 11-5	Returns string for a location.

XmlXPtrLocGetNode

Returns node from location

Syntax

```
xmlnode* XmlXPtrLocGetNode(
   xmlxptrloc* loc);
```

Parameter	In/Out	Description
`loc`	`IN`	location

Returns

`(xmlnode *) Node` from location

XmlXPtrLocGetPoint

Returns point from location

Syntax

```
xmlpoint* XmlXPtrLocGetPoint(
```

```
   xmlxptrloc* loc);
```

Parameter	In/Out	Description
loc	IN	location

Returns

(xmlpoint *) Point from location

XmlXPtrLocGetRange

Returns range from location.

Syntax

```
xmlrange* XmlXPtrLocGetRange(
   xmlxptrloc* loc);
```

Parameter	In/Out	Description
loc	IN	location

Returns

(xmlrange *) Range from location

XmlXPtrLocGetType

Returns type of location

Syntax

```
xmlxptrloctype XmlXPtrLocGetType(
   xmlxptrloc* loc);
```

Parameter	In/Out	Description
loc	IN	location

Returns

`(xmlxptrloctype)` Type of location

XmlXPtrLocToString

Returns string for a location:

- node name: name of the container node

- names of container nodes: "not a location" otherwise

Syntax

```
oratext* XmlXPtrLocToString(
   xmlxptrloc* loc);
```

Parameter	In/Out	Description
`loc`	`IN`	location

Returns

`(oratext *)` string

XPtrLocSet Interface

Table 11–3 summarizes the methods of available through the `XPtrLocSet` interface.

Table 11–3 Summary of XPtrLocSet Methods; Package XPointer

Function	Summary
XmlXPtrLocSetFree on page 11-6	Free a location set
XmlXPtrLocSetGetItem on page 11-6	Returns location with `idx` position in `XPtrLocSet`
XmlXPtrLocSetGetLength on page 11-7	Returns length of `XPtrLocSet`.

XmlXPtrLocSetFree

It is user's responsibility to call this function on every location set returned by `XPointer` or `XPtrLocSet` interfaces

Syntax

```
void XmlXPtrLocSetFree(
   xmlxptrlocset* locset);
```

Parameter	In/Out	Description
`locset`	`IN`	location set

XmlXPtrLocSetGetItem

Returns location with idx position in the location set. First position is `1`.

Syntax

```
xmlxptrloc* XmlXPtrLocSetGetItem(
   xmlxptrlocset* locset,
   ub4 idx);
```

Parameter	In/Out	Description
locset	IN	location set
idx	IN	location index

Returns

(xmlxptrloc *) location with the position idx

XmlXPtrLocSetGetLength

Returns the number of locations in the location set

Syntax

```
ub4 XmlXPtrLocSetGetLength(
   xmlxptrlocset* locset);
```

Parameter	In/Out	Description
locset	IN	location set

Returns

(ub4) number of nodes in locset

12

Package XSLT APIs for C

Package XSLT implements types and methods related to XSL processing.

This chapter contains this section:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XSLT Interface

Table 12–1 summarizes the methods of available through the XSLT interface.

Table 12–1 Summary of XSLT Methods

Function	Summary
XmlXslCreate on page 12-2	Create an XSL context.
XmlXslDestroy on page 12-3	Destroy an XSL context.
XmlXslGetBaseURI on page 12-3	Get the XSL processor base URI.
XmlXslGetOutput on page 12-4	Get the XSL result fragment.
XmlXslGetStylesheetDom on page 12-4	Get he XSL stylesheet document.
XmlXslGetTextParam on page 12-5	Get the XSL text parameter value.
XmlXslProcess on page 12-5	Perform XSL processing on an instance document.
XmlXslResetAllParams on page 12-6	Reset XSL processor parameters.
XmlXslSetOutputDom on page 12-6	Set the XSL context output DOM.
XmlXslSetOutputEncoding on page 12-7	Set the XSL context output encoding.
XmlXslSetOutputMethod on page 12-7	Set the XSL context output method.
XmlXslSetOutputSax on page 12-8	Set the XSL context output SAX.
XmlXslSetOutputStream on page 12-8	Set the XSL context output stream.
XmlXslSetTextParam on page 12-9	Set the XSL context output text parameter.

XmlXslCreate

Create an XSLT context

Syntax

```
xslctx *XmlXslCreate(
   xmlctx *ctx,
   xmldocnode *xsl,
   oratext *baseuri,
   xmlerr *err);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
xsl	IN	XSL stylesheet document object
baseuri	IN	base URI for including and importing documents
err	IN/OUT	returned error code

Returns

(xslctx *) XSLT context

See Also: XmlXslDestroy

XmlXslDestroy

Destroy an XSL context

Syntax

```
xmlerr XmlXslDestroy(
   xslctx *ctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context

Returns

(xmlerr) error code

See Also: XmlXslCreate

XmlXslGetBaseURI

Get the XSL processor base URI

Syntax

```
oratext *XmlXslGetBaseURI(
   xslctx *ctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context object

Returns

`(oratext *)` base URI

XmlXslGetOutput

Get the XSL result fragment

Syntax

```
xmlfragnode *XmlXslGetOutput(
   xslctx *ctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context object

Returns

`(xmlfragnode *)` result fragment

XmlXslGetStylesheetDom

Get the XSL stylesheet document

Syntax

```
xmldocnode *XmlXslGetStylesheetDom(
   xslctx *ctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context object

Returns

(xmldocnode *) stylesheet document

XmlXslGetTextParam

Get the XSL text parameter value

Syntax

```
oratext *XmlXslGetTextParam(
   xslctx *ctx,
   oratext *name);
```

Parameter	In/Out	Description
ctx	IN	XML context object
name	IN	name of the top-level parameter value

Returns

(oratext *) parameter value

See Also: XmlXslSetTextParam

XmlXslProcess

Do XSL processing on an instance document

Syntax

```
xmlerr XmlXslProcess(
   xslctx *ctx,
   xmldocnode *xml,
   boolean normalize);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
xml	IN	instance document to process
normalize	IN	if TRUE, force the XSL processor to normalize the document

Returns

(xmlerr) error code

XmlXslResetAllParams

Reset all the top level parameters added

Syntax

```
xmlerr XmlXslResetAllParams(
   xslctx *ctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context object

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

See Also: XmlXslSetTextParam

XmlXslSetOutputDom

Set the xslctx output DOM

Syntax

```
xmlerr XmlXslSetOutputDom(
   xslctx *ctx,
   xmldocnode *doc);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
doc	IN	output node

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

XmlXslSetOutputEncoding

Set the xslctx output encoding

Syntax

```
xmlerr XmlXslSetOutputEncoding(
   xslctx *ctx,
   oratext* encoding);
```

Parameter	In/Out	Description
ctx	IN	XML context object
encoding	IN	output encoding

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

XmlXslSetOutputMethod

Set the xslctx output method

Syntax

```
xmlerr XmlXslSetOutputMethod(
   xslctx *ctx,
   xmlxslomethod method);
```

Parameter	In/Out	Description
ctx	IN	XML context object
encoding	IN	XSL output method

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

XmlXslSetOutputSax

Set the xslctx output SAX

Syntax

```
xmlerr XmlXslSetOutputSax(
   xslctx *ctx,
   xmlsaxcb* saxcb,
   void *saxctx);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
saxcb	IN	SAX callback object
saxctx	IN	SAX callback context

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

XmlXslSetOutputStream

Syntax

```
xmlerr XmlXslSetOutputStream(
   xslctx *ctx,
   xmlostream *stream);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
stream	IN	output stream object

Returns

(xmlxsl) error code, XMLXSL_SUCC [0] on success.

XmlXslSetTextParam

Set the xslctx output text parameter.

Syntax

```
xmlerr XmlXslSetTextParam(
   xslctx *ctx,
   oratext *name,
   oratext *value);
```

Parameter	In/Out	Description
ctx	IN	XSL context object
name	IN	name of top level parameter
value	IN	value of top level parameter

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

See Also: XmlXslGetTextParam

13

Package XSLTVM APIs for C

Package XSLTVM implements the XSL Transformation (XSLT) language as specified in W3C Recommendation 16 November 1999. The XSLTVM package contains two interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Using XSLTVM

XSLT Virtual Machine is the software implementation of a "CPU" designed to run compiled XSLT code. A concept of virtual machine assumes a compiler compiling XSLT stylesheets to a sequence of byte codes or machine instructions for the "XSLT CPU". The byte-code program is a platform-independent sequence of 2-byte units. It can be stored, cached and run on different XSLTVM. The XSLTVM uses the bytecode programs to transform instance XML documents. This approach clearly separates compile (design)-time from run-time computations and specifies a uniform way of exchanging data between instructions.

A typical scenario of using the package APIs has the following steps:

1. Create/Use an XML meta context object.

   ```
   xctx = XmlCreate(,...);
   ```

2. Create/Use an XSLT Compiler object.

   ```
   comp = XmlXvmCreateComp(xctx);
   ```

3. Compile an XSLT stylesheets and cache the result bytecode.

   ```
   code = XmlXvmCompileFile(comp, xslFile, baseuri, flags, );
   ```

4. Create/Use an XSLTVM object. The explicit stack size setting are needed when XSLTVM terminates with "... Stack Overflow" message or when smaller memory footprints are required (see `XmlXvmCreate`).

   ```
   vm = XmlXvmCreate(xctx, "StringStack", 32, "NodeStack", 24,
   NULL);
   ```

5. Set a stylesheet bytecode to the XSLTVM object.

   ```
   len = XmlXvmGetBytecodeLength(code, ); err =
   XmlXvmSetBytecodeBuffer(vm, code, len);
   ```

6. Transform an instance XML document.

   ```
   err = XmlXvmTransformFile(vm, xmlFile, baseuri);
   ```

7. Clean.

   ```
   XmlXvmDestroy(vm);

   XmlXvmDestroyComp(comp);

   XmlDestroy(xctx);
   ```

XSLTC Interface

Table 13–1 summarizes the methods of available through the XSLTVM Interface.

Table 13–1 Summary of XSLTC Methods; XSLTVM Package

Function	Summary
XmlXvmCompileBuffer on page 13-3	Compile an XSLT stylesheet from buffer into bytecode.
XmlXvmCompileDom on page 13-4	Compile an XSLT stylesheet from DOM into bytecode.
XmlXvmCompileFile on page 13-5	Compile an XSLT stylesheet from file into bytecode.
XmlXvmCompileURI on page 13-6	Compile XSLT stylesheet from URI into byte code.
XmlXvmCompileXPath on page 13-7	Compile an XPath expression.
XmlXvmCreateComp on page 13-8	Create an XSLT compiler.
XmlXvmDestroyComp on page 13-8	Destroy an XSLT compiler object.
XmlXvmGetBytecodeLength on page 13-9	Returns the bytecode length.

XmlXvmCompileBuffer

Compile an XSLT stylesheet from buffer into bytecode. Compiler flags could be one or more of the following:

- `XMLXVM_DEBUG` forces compiler to include debug information into the bytecode
- `XMLXVM_STRIPSPACE` is equivalent to `<xsl:strip-space elements="*"/>`.

The generated bytecode resides in a compiler buffer which is freed when next stylesheet is compiled or when compiler object is deleted. Hence, if the bytecode is to be reused it should be copied into another location.

Syntax

```
ub2 *XmlXvmCompileBuffer(
   xmlxvmcomp *comp,
   oratext *buffer,
```

```
   ub4 length,
   oratext *baseURI,
   xmlxvmflag flags,
   xmlerr *error);
```

Parameter	In/Out	Description
comp	IN	compiler object
buffer	IN	pointer to buffer containing stylesheet document
length	IN	length of the stylesheet document in bytes
baseuri	IN	base URI of the document
flags	IN	flags for the current compilation
error	OUT	returned error code

Returns

(ub2 *) bytecode or NULL on error

> **See Also:** XmlXvmCompileFile, XmlXvmCompileURI, XmlXvmCompileDom

XmlXvmCompileDom

Compile an XSLT stylesheet from DOM into bytecode. Compiler flags could be one or more of the following:

- XMLXVM_DEBUG forces compiler to include debug information into the bytecode
- XMLXVM_STRIPSPACE is equivalent to <xsl:strip-space elements="*"/>.

The generated bytecode resides in a compiler buffer which is freed when next stylesheet is compiled or when compiler object is deleted. Hence, if the bytecode is to be reused it should be copied into another location.

Syntax

```
ub2 *XmlXvmCompileDom(
   xmlxvmcomp *comp,
   xmldocnode *root,
   xmlxvmflag flags,
```

```
   xmlerr *error);
```

Parameter	In/Out	Description
comp	IN	compiler object
rooot	IN	root element of the stylesheet DOM
flags	IN	flags for the current compilation
error	OUT	returned error code

Returns

(ub2 *) bytecode or NULL on error

See Also: XmlXvmCompileFile, XmlXvmCompileBuffer, XmlXvmCompileURI

XmlXvmCompileFile

Compile XSLT stylesheet from file into bytecode. Compiler flags could be one or more of the following:

- XMLXVM_DEBUG forces compiler to include debug information into the bytecode
- XMLXVM_STRIPSPACE is equivalent to <xsl:strip-space elements="*"/>.

The generated bytecode resides in a compiler buffer which is freed when next stylesheet is compiled or when compiler object is deleted. Hence, if the bytecode is to be reused it should be copied into another location.

Syntax

```
ub2 *XmlXvmCompileFile(
   xmlxvmcomp *comp,
   oratext *path,
   oratext *baseURI,
   xmlxvmflag flags,
   xmlerr *error);
```

Parameter	In/Out	Description
comp	IN	compiler object
path	IN	path of XSL stylesheet file
baseuri	IN	base URI of the document
flags	IN	flags for the current compilation
error	OUT	returned error code

Returns

(ub2 *) bytecode or NULL on error

See Also: XmlXvmCompileURI, XmlXvmCompileBuffer, XmlXvmCompileDom

XmlXvmCompileURI

Compile XSLT stylesheet from URI into bytecode. Compiler flags could be one or more of the following:

- XMLXVM_DEBUG forces compiler to include debug information into the bytecode
- XMLXVM_STRIPSPACE is equivalent to <xsl:strip-space elements="*"/>.

The generated bytecode resides in a compiler buffer which is freed when next stylesheet is compiled or when compiler object is deleted. Hence, if the bytecode is to be reused it should be copied into another location.

Syntax

```
ub2 *XmlXvmCompileURI(
   xmlxvmcomp *comp,
   oratext *uri,
   xmlxvmflag flags,
   xmlerr *error);
```

Parameter	In/Out	Description
comp	IN	compiler object

Parameter	In/Out	Description
uri	IN	URI of the file that contains the XSL stylesheet
flags	IN	flags for the current compilation
error	OUT	returned error code

Returns

(ub2 *) bytecode or NULL on error

See Also: XmlXvmCompileFile, XmlXvmCompileBuffer, XmlXvmCompileDom

XmlXvmCompileXPath

Compiles an XPath expression. The optional pfxmap is used to map namespace prefixes to URIs in the XPath expression. It is an array of prefix, URI values, ending in NULL, and so on.

Syntax

```
ub2 *XmlXvmCompileXPath(
   xmlxvmcomp *comp,
   oratext *xpath,
   oratext **pfxmap,
   xmlerr *error);
```

Parameter	In/Out	Description
comp	IN	compiler object
xpath	IN	XPath expression
pfxmap	IN	array of prefix-URI mappings
error	OUT	returned error code

Returns

(ub2 *) XPath expression bytecode or NULL on error

XmlXvmCreateComp

Create an XSLT compiler object. The XSLT compiler is used to compile XSLT stylesheets into bytecode.

Syntax

```
xmlxvmcomp *XmlXvmCreateComp(
   xmlctx *xctx);
```

Parameter	In/Out	Description
xctx	IN	XML context

Returns

`(xmlxvmcomp *)` XSLT compiler object, or `NULL` on error

See Also: XmlXvmDestroyComp

XmlXvmDestroyComp

Destroys an XSLT compiler object

Syntax

```
void XmlXvmDestroyComp(
   xmlxvmcomp *comp);
```

Parameter	In/Out	Description
comp	IN	XSLT compiler object

See Also: XmlXvmCreateComp

XmlXvmGetBytecodeLength

The bytecode length is needed when the bytecode is to be copied or when it is set into XSLTVM.

Syntax

```
ub4 XmlXvmGetBytecodeLength(
   ub2 *bytecode,
   xmlerr *error);
```

Parameter	In/Out	Description
bytecode	IN	bytecode buffer
error	OUT	returned error code

Returns

(ub4) The bytecode length in bytes.

XSLTVM Interface

Table 13–2 summarizes the methods of available through the XSLTVM Interface.

Table 13–2 Summary of XSLTVM Methods; Package XSLTVM

Function	Summary
XMLXVM_DEBUG_F on page 13-11	XMLXSLTVM debug function.
XmlXvmCreate on page 13-12	Create an XSLT virtual machine.
XmlXvmDestroy on page 13-12	Destroys an XSLT virtual machine.
XmlXvmEvaluateXPath on page 13-13	Evaluate already-compiled XPath expression.
XmlXvmGetObjectBoolean on page 13-13	Get boolean value of XPath object.
XmlXvmGetObjectNSetNode on page 13-14	Get node from nodeset type XPathobject.
XmlXvmGetObjectNSetNum on page 13-15	Get number of nodes in nodeset type XPathobject.
XmlXvmGetObjectNumber on page 13-15	Get number from XPath object.
XmlXvmGetObjectString on page 13-16	Get string from XPath object.
XmlXvmGetObjectType on page 13-16	Get XPath object type.
XmlXvmGetOutputDom on page 13-17	Returns the output DOM.
XmlXvmResetParams on page 13-17	Resets the stylesheet top level text parameters.
XmlXvmSetBaseURI on page 13-18	Sets the base URI for the XLTVM.
XmlXvmSetBytecodeBuffer on page 13-18	Set the compiled bytecode.
XmlXvmSetBytecodeFile on page 13-19	Set the compiled byte code from file.
XmlXvmSetBytecodeURI on page 13-19	Set the compiled bytecode.
XmlXvmSetDebugFunc on page 13-20	Set a callback function for debugging.
XmlXvmSetOutputDom on page 13-21	Sets the XSLTVM to output document node.
XmlXvmSetOutputEncoding on page 13-21	Sets the encoding for the XSLTVM output.
XmlXvmSetOutputSax on page 13-22	Sets XSLTVM to output SAX.
XmlXvmSetOutputStream on page 13-22	Set the XSLTVM output to a user-defined stream.
XmlXvmSetTextParam on page 13-23	Set the stylesheet top-level text parameter.

Table 13–2 (Cont.) Summary of XSLTVM Methods; Package XSLTVM

Function	Summary
XmlXvmTransformBuffer on page 13-23	Run compiled XSLT stylesheet on XML document in memory.
XmlXvmTransformDom on page 13-24	Run compiled XSLT stylesheet on XML document as DOM.
XmlXvmTransformFile on page 13-25	Run compiled XSLT stylesheet on XML document in file.
XmlXvmTransformURI on page 13-25	Run compiled XSLT stylesheet on XML document from URI.

XMLXVM_DEBUG_F

Debug callback function for XSLT VM

Syntax

```
#define XMLXVM_DEBUG_F(func, line, file, obj, n)
void func(
   ub2 line,
   oratext *file,
   xvmobj *obj,
   ub4 n)
```

Parameter	In/Out	Description
line	IN	source stylesheet line number
file	IN	stylesheet filename
obj	IN	current VM object
n	IN	index of current node

See Also: XmlXvmSetDebugFunc

XmlXvmCreate

Create an XSLT virtual machine. Zero or more of the following XSLTVM properties could be set by using this API:

- `"VMStack", size` sets the size[Kbyte] of the main VM stack; default size is 4K.
- `"NodeStack", size` sets the size[Kbyte] of the node-stack; default size is 16K.
- `"StringStack", size` sets the size[Kbyte] of the string-stack; default size is 64K.

If the stack size is not specified the default size is used. The explicit stack size setting is needed when XSLTVM terminates with "Stack Overflow" message or when smaller memory footprints are required.

Syntax

```
xmlxvm *XmlXvmCreate(
   xmlctx *xctx,
   list);
```

Parameter	In/Out	Description
`xctx`	`IN`	XML context
`list`	`IN`	`NULL`-terminated list of properties to set; can be empty

Returns

`(xmlxvm *)` XSLT virtual machine object, or `NULL` on error

See Also: XmlXvmDestroy

XmlXvmDestroy

Destroys an XSLT virtual machine

Syntax

```
void XmlXvmDestroy(
```

```
   xmlxvm *vm);
```

Parameter	In/Out	Description
vm	IN	VM object

See Also: XmlXvmCreate

XmlXvmEvaluateXPath

Evaluate already-compiled XPath expression

Syntax

```
xvmobj *XmlXvmEvaluateXPath(
   xmlxvm *vm,
   ub2 *bytecode,
   ub4 ctxpos,
   ub4 ctxsize,
   xmlnode *ctxnode);
```

Parameter	In/Out	Description
vm	IN	XSLTVM object
bytecode	IN	XPath expression bytecode
ctxpos	IN	current context position
ctxsize	IN	current context size
ctxnode	IN	current context node

Returns

`(xvmobj *)` XPath object

XmlXvmGetObjectBoolean

Get boolean value of XPath object

Syntax

```
boolean XmlXvmGetObjectBoolean(
   xvmobj *obj);
```

Parameter	In/Out	Description
obj	IN	object

Returns

`(boolean)` value of an XPath object

See Also: XmlXvmGetObjectType, XmlXvmGetObjectNSetNum, XmlXvmGetObjectNSetNode, XmlXvmGetObjectNumber, XmlXvmGetObjectBoolean

XmlXvmGetObjectNSetNode

Get node from nodeset-type XPath object

Syntax

```
xmlnode *XmlXvmGetObjectNSetNode(
   xvmobj *obj,
   ub4 i);
```

Parameter	In/Out	Description
obj	IN	object
i	IN	node index in nodeset

Returns

`(xmlnode *)` The object type or values.

See Also: XmlXvmGetObjectType, XmlXvmGetObjectNSetNum, XmlXvmGetObjectString, XmlXvmGetObjectNumber, XmlXvmGetObjectBoolean

XmlXvmGetObjectNSetNum

Get number of nodes in nodeset-type XPath object

Syntax

```
ub4 XmlXvmGetObjectNSetNum(
   xvmobj *obj);
```

Parameter	In/Out	Description
obj	IN	object

Returns

(ub4) number of nodes in nodeset

See Also: XmlXvmGetObjectType, XmlXvmGetObjectNSetNode, XmlXvmGetObjectString, XmlXvmGetObjectNumber, XmlXvmGetObjectBoolean

XmlXvmGetObjectNumber

Get number from XPath object.

Syntax

```
double XmlXvmGetObjectNumber(
   xvmobj *obj);
```

Parameter	In/Out	Description
obj	IN	object

Returns

(double) number

See Also: XmlXvmGetObjectType, XmlXvmGetObjectNSetNum, XmlXvmGetObjectNSetNode, XmlXvmGetObjectString, XmlXvmGetObjectBoolean

XmlXvmGetObjectString

Get string from XPath object.

Syntax

```
oratext *XmlXvmGetObjectString(
   xvmobj *obj);
```

Parameter	In/Out	Description
obj	IN	object

Returns

`(oratext *)` string

See Also: XmlXvmGetObjectType, XmlXvmGetObjectNSetNum, XmlXvmGetObjectNSetNode, XmlXvmGetObjectNumber, XmlXvmGetObjectBoolean

XmlXvmGetObjectType

Get XPath object type

Syntax

```
xmlxvmobjtype XmlXvmGetObjectType(
   xvmobj *obj);
```

Parameter	In/Out	Description
obj	IN	object

Returns

`(xmlxvmobjtype)` type-code for object

See Also: XmlXvmGetObjectNSetNum, XmlXvmGetObjectNSetNode, XmlXvmGetObjectString, XmlXvmGetObjectNumber, XmlXvmGetObjectBoolean

XmlXvmGetOutputDom

Returns the root node of the result DOM tree (if any). `XmlXvmSetOutputDom` has to be used before transformation to set the VM to output a DOM tree (the default VM output is a stream).

Syntax

```
xmlfragnode *XmlXvmGetOutputDom(
   xmlxvm *vm);
```

Parameter	In/Out	Description
`vm`	`IN`	VM object

Returns

`(xmlfragnode *)` output DOM, or `NULL` in a case of SAX or Stream output.

See Also: XmlXvmSetOutputDom

XmlXvmResetParams

Resets the stylesheet top-level parameters with their default values.

Syntax

```
void XmlXvmResetParams(
   xmlxvm *vm);
```

Parameter	In/Out	Description
vm	IN	VM object

XmlXvmSetBaseURI

Sets the base URI for the XSLTVM. The baseuri is used by VM to the compose the path XML documents to be loaded for transformation using `document` or `XmlXvmTransformFile`.

Syntax

```
xmlerr XmlXvmSetBaseURI(
   xmlxvm *vm,
   oratext *baseuri);
```

Parameter	In/Out	Description
vm	IN	VM object
baseuri	IN	VM base URI for reading and writing documents

Returns

`(xmlerr)` error code.

XmlXvmSetBytecodeBuffer

Set the compiled bytecode from buffer. Any previously set bytecode is replaced. An XML transformation can't be performed if the stylesheet bytecode is not set. The VM doesn't copy the bytecode into internal buffer, hence the it shouldn't be freed before VM finishes using it.

Syntax

```
xmlerr XmlXvmSetBytecodeBuffer(
   xmlxvm *vm,
   ub2 *buffer,
   size_t buflen);
```

Parameter	In/Out	Description
vm	IN	XSLT VM context
buffer	IN	user's buffer
buflen	IN	size of buffer, in bytes

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlXvmSetBytecodeFile, XmlXvmSetBytecodeURI

XmlXvmSetBytecodeFile

Set the compiled bytecode from file. Any previously set bytecode is replaced. An XML transformation can't be performed if the stylesheet bytecode is not set.

Syntax

```
xmlerr XmlXvmSetBytecodeFile(
   xmlxvm *vm,
   oratext *path);
```

Parameter	In/Out	Description
vm	IN	XSLT VM context
path	IN	path of bytecode file

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlXvmSetBytecodeURI, XmlXvmSetBytecodeBuffer

XmlXvmSetBytecodeURI

Set the compiled bytecode from URI. Any previously set bytecode is replaced. An XML transformation can't be performed if the stylesheet bytecode is not set.

Syntax

```
xmlerr XmlXvmSetBytecodeURI(
   xmlxvm *vm,
   oratext *uri);
```

Parameter	In/Out	Description
vm	IN	XSLT VM context
uri	IN	path of bytecode file

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

See Also: XmlXvmSetBytecodeFile, XmlXvmSetBytecodeBuffer

XmlXvmSetDebugFunc

The user callback function is invoked by VM every time the execution reaches a new line in the XSLT stylesheet. The VM passes to the user the stylesheet file name, the line number, the current context nodes-set and the current node index in the node-set. IMPORTANT - the stylesheet has to be compiled with flag XMLXVM_DEBUG.

Syntax

```
#define XMLXVM_DEBUG_FUNC(func)
void func (ub2 line, oratext *filename, xvmobj *obj, ub4 n)
xmlerr XmlXvmSetDebugFunc(
   xmlxvm *vm,
   XMLXVM_DEBUG_FUNC(debugcallback));
```

Parameter	In/Out	Description
vm	IN	XSLT VM context
func	IN	callback function

Returns

(xmlerr) numeric error code, XMLERR_OK [0] on success

XmlXvmSetOutputDom

Sets the XSLTVM to output DOM. If `xmldocnode==NULL`, then the result DOM tree belongs to the VM object and is deleted when a new transformation is performed or when the VM object is deleted. If the result DOM tree is to be used for longer period of time then an `xmldocnode` has to be created and set to the VM object.

Syntax

```
xmlerr XmlXvmSetOutputDom(
   xmlxvm *vm,
   xmldocnode *doc);
```

Parameter	In/Out	Description
`vm`	`IN`	VM object
`doc`	`IN`	empty document

Returns

`(xmlerr)` error code

XmlXvmSetOutputEncoding

Sets the encoding for the XSLTVM stream output. If the input (data) encoding is different from the one set by this APIs then encoding conversion is performed. This APIs overrides the encoding set in the XSLT stylesheet (if any).

Syntax

```
xmlerr XmlXvmSetOutputEncoding(
   xmlxvm *vm,
   oratext *encoding);
```

Parameter	In/Out	Description
`vm`	`IN`	VM object
`encoding`	`IN`	output encoding

Returns

(xmlerr) error code.

XmlXvmSetOutputSax

Set XSLTVM to output SAX. If the SAX callback interface object is provided the VM outputs the result document in a form of SAX events using the user specified callback functions.

Syntax

```
xmlerr XmlXvmSetOutputSax(
   xmlxvm *vm,
   xmlsaxcb *saxcb,
   void *saxctx);
```

Parameter	In/Out	Description
vm	IN	VM object
saxcb	IN	SAX callback object
saxctx	IN	SAX context

Returns

(xmlerr) error code

XmlXvmSetOutputStream

Set the XSLTVM output to a user-defined stream. The default XSLTVM output is a stream. This APIs overrides the default stream with user specified APIs for writing.

Syntax

```
xmlerr XmlXvmSetOutputStream(
   xmlxvm *vm,
   xmlostream *ostream);
```

Parameter	In/Out	Description
vm	IN	VM object

Parameter	In/Out	Description
ostream	IN	stream object

Returns

(xmlerr) error code.

XmlXvmSetTextParam

Set the stylesheet top-level text parameter. The parameter value set in the XSLT stylesheet is overwritten. Since the top-level parameters are reset with stylesheet values after each transformation, this APIs has to be called again.

Syntax

```
xmlerr XmlXvmSetTextParam(
   xmlxvm *vm,
   oratext *name,
   oratext *value);
```

Parameter	In/Out	Description
vm	IN	VM object
name	IN	name of top-level parameter
value	IN	value of top-level parameter

Returns

(xmlerr) error code, XMLERR_SUCC [0] on success.

XmlXvmTransformBuffer

Run compiled XSLT stylesheet on XML document in memory. The compiled XSLT stylesheet (bytecode) should be set using XmlXvmSetBytecodeXXX prior to this call.

Syntax

```
xmlerr XmlXvmTransformBuffer(
   xmlxvm *vm,
```

```
   oratext *buffer,
   ub4 length,
   oratext *baseURI);
```

Parameter	In/Out	Description
vm	IN	VM object
buffer	IN	NULL-terminated buffer that contains the XML document
length	IN	length of the XML document
baseURI	IN	base URI of XML document

Returns

(xmlerr) error code.

See Also: XmlXvmTransformFile, XmlXvmTransformURI, XmlXvmTransformDom

XmlXvmTransformDom

Run compiled XSLT stylesheet on XML document as DOM. The compiled XSLT stylesheet (bytecode) should be set using XmlXvmSetBytecodeXXX prior to this call.

Syntax

```
xmlerr XmlXvmTransformDom(
   xmlxvm *vm,
   xmldocnode *root);
```

Parameter	In/Out	Description
vm	IN	VM object
root	IN	root element of XML document's DOM

Returns

(xmlerr) error code.

See Also: XmlXvmTransformFile, XmlXvmTransformURI, XmlXvmTransformBuffer

XmlXvmTransformFile

Run compiled XSLT stylesheet on XML document in file. The compiled XSLT stylesheet (bytecode) should be set using `XmlXvmSetBytecodeXXX` prior to this call.

Syntax

```
xmlerr XmlXvmTransformFile(
   xmlxvm *vm,
   oratext *path,
   oratext *baseURI);
```

Parameter	In/Out	Description
`vm`	`IN`	VM object
`path`	`IN`	path of XML document to transform
`baseURI`	`IN`	base URI of XML document

Returns

`(xmlerr)` error code

See Also: XmlXvmTransformURI, XmlXvmTransformBuffer, XmlXvmTransformDom

XmlXvmTransformURI

Run compiled XSLT stylesheet on XML document from URI. The compiled XSLT stylesheet (bytecode) should be set using `XmlXvmSetBytecodeXXX` prior to this call.

Syntax

```
xmlerr XmlXvmTransformURI(
   xmlxvm *vm,
```

```
    oratext *uri);
```

Parameter	In/Out	Description
vm	IN	VM object
uri	IN	URI of XML document to transform

Returns

(xmlerr) error code.

See Also: XmlXvmTransformFile, XmlXvmTransformBuffer, XmlXvmTransformDom

Part II

XML APIs for C++

This part contains the following chapters:

14

Package Ctx APIs for C++

Ctx is the namespace for TCtx XML context related types and interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Ctx Datatypes

Table 14–1 summarizes the datatypes of the Ctx package.

Table 14–1 Summary of Datatypes; Ctx Package

Datatype	Description
encoding on page 14-2	A single supported encoding.
encodings on page 14-2	An array of encodings.

encoding

A single supported encoding.

Definition

```
typedef struct encoding {
   oratext *encname;
   oratext *encvalue;
} encoding;
```

encodings

An array of encodings.

Definition

```
typedef struct encodings {
   unsigned num;
   encoding *enc;
} encodings;
```

MemAllocator Interface

Table 14–2 summarizes the methods of available through the `MemAllocator` interface.

Table 14–2 Summary of MemAllocator Methods; Ctx Package

Function	Summary
alloc on page 14-3	Allocates memory of given size.
dealloc on page 14-3	Deallocate memory pointed to by the argument.
~MemAllocator on page 14-4	Virtual destructor - interface level handle to actual destructors.

alloc

This is a virtual member function that defines a prototype for user defined allocator functions

Syntax

```
virtual void* alloc(
   ub4 size) = 0;
```

Parameter	Description
`size`	memory size

dealloc

This is a virtual member function that defines a prototype for user defined deallocator functions. Such deallocators are supposed to deallocate memory allocated by the alloc member functions

Syntax

```
virtual void dealloc(
   void* ptr) = 0;
```

Parameter	Description
ptr	pointer to previously allocated memory

~MemAllocator

It provides an interface level handle to actual destructors that can be invoked without knowing their names or implementations

Syntax

```
virtual ~MemAllocator() {}
```

TCtx Interface

Table 14–3 summarizes the methods of available through the `TCtx` interface.

Table 14–3 Summary of TCtx Methods; Ctx Package

Function	Summary
TCtx on page 14-5	Class constructor.
getEncoding on page 14-6	Get data encoding in use by XML context.
getErrHandler on page 14-6	Get Error Handler provided by the user.
getMemAllocator on page 14-7	Get memory allocator.
isSimple on page 14-7	Get a flag that indicates if data encoding is simple.
isUnicode on page 14-7	Get a flag indicating if data encoding is Unicode.
~TCtx on page 14-8	Destructor - clears space and destroys the implementation.

TCtx

TCtx constructor. It throws `XmlException` if it fails to create a context object.

Syntax	Description
`TCtx() throw (XMLException)`	This constructor creates the context object and initializes it with default values of parameters.

`TCtx(`
`  oratext* name,`
`  ErrorHandler* errh = NULL,`
`  MemAllocator* memalloc = NULL,`
`  encodings* encs = NULL)`
`throw (XMLException)`
	This constructor creates the context object and initializes it with parameter values provided by the user.

`TCtx(`
`  oratext* name,`
`  up4 inpblksize,`
`  ErrorIfs* errh = NULL,`
`  MemAllocator* memalloc = NULL,`
`  encodings* encs = NULL)`
`throw (XMLException)`
	This constructor creates the context object and initializes it with parameter values provided by the user. Takes an additional parameter for memory block size from input source.

Parameter	Description
name	user defined name of the context
errh	user defined error handler
memalloc	user defined memory allocator
encs	user specified encodings
inpblksize	memory block size for input source

Returns

(TCtx) Context object

getEncoding

Returns data encoding in use by XML context. Ordinarily, the data encoding is chosen by the user, so this function is not needed. However, if the data encoding is not specified, and allowed to default, this function can be used to return the name of that default encoding.

Syntax

```
oratext* getEncoding() const;
```

Returns

(oratext *) name of data encoding

getErrHandler

This member functions returns Error Handler provided by the user when the context was created, or NULL if none were provided.

Syntax

```
ErrorHandler* getErrHandler() const;
```

Returns

`(ErrorHandler *)` Pointer to the Error Handler object, or `NULL`

getMemAllocator

This member function returns memory allocator provided by the user when the context was created, or default memory allocator. It is important that this memory allocator is used for all C level memory allocations

Syntax

```
MemAllocator* getMemAllocator() const;
```

Returns

`(MemAllocator*)` Pointer to the memory allocator object

isSimple

Returns a flag saying whether the context's data encoding is "simple", single-byte for each character, like ASCII or EBCDIC.

Syntax

```
boolean isSimple() const;
```

Returns

`(boolean) TRUE` of data encoding is "simple", `FALSE` otherwise

isUnicode

Returns a flag saying whether the context's data encoding is Unicode, UTF-16, with two-byte for each character.

Syntax

```
boolean isUnicode() const;
```

Returns

`(boolean) TRUE` if data encoding is Unicode, `FALSE` otherwise

~TCtx

Destructor - should be called by the user the context object is no longer needed

Syntax

```
~Tctx();
```

15

Package Dom APIs for C++

Interfaces in this package represent DOM level 2 Core interfaces according to `http://www.w3c.org/TR/DOM-Level-2-Core/core.html`.

This chapter contains the following sections:

- NamedNodeMapRef Interface
- NodeFilter Interface
- NodeIterator Interface
- NodeListRef Interface
- NodeRef Interface
- NotationRef Interface
- ProcessingInstructionRef Interface
- Range Interface
- RangeException Interface
- TextRef Interface
- TreeWalker Interface

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Using Dom

DOM interfaces are represented as generic references to different implementations of the DOM specifications. They are parameterized by Node, which supports various specializations and instantiations. Of them, the most important is `xmlnode` that corresponds to the current C implementation.

These generic references do not have a `NULL`-like value. Any implementation should never create a stateless reference. If there is need to signal that something has no state, an exception should be thrown.

Many methods might throw the `SYNTAX_ERR` exception, if the DOM tree is incorrectly formed, or `UNDEFINED_ERR`, in the case of wrong parameters or unexpected `NULL` pointers. If these are the only errors that a particular method might throw, it is not reflected in the method signature.

Actual DOM trees do not dependent on the context (`TCtx`). However, manipulations on DOM trees in the current, `xmlctx` based implementation require access to the current context (`TCtx`). This is accomplished by passing the context pointer to the constructor of DOMImplRef. In multithreaded environment `DOMImplRef` is always created in the thread context and, so, has the pointer to the right context.

`DOMImplRef` providea a way to create DOM trees. DomImplRef is a reference to the actual `DOMImplementation` object that is created when a regular, non-copy constructor of `DomImplRef` is invoked. This works well in multithreaded environment where DOM trees need to be shared, and each thread has a separate TCtx associated with it. This works equally well in a single threaded environment.

`DOMString` is only one of encodings supported by Oracle implementations. The support of other encodings is Oracle's extension. The `oratext*` data type is used for all encodings.

Dom Datatypes

Table 15–1 summarizes the datatypes of the Dom package.

Table 15–1 Summary of Datatypes; Dom Package

Datatype	Description
AcceptNodeCodes on page 15-4	Defines values returned by node filters.
CompareHowCode on page 15-4	Defines type of comparison.
DOMNodeType on page 15-5	Defines type of node.
DOMExceptionCode on page 15-5	Defines codes for DOM exception.
WhatToShowCode on page 15-6	Defines codes for filtering.
RangeExceptionCode on page 15-6	Codes for DOM Range exceptions.

AcceptNodeCodes

Defines values returned by node filters. Used by node iterators and tree walkers.

Definition

```
typedef enum AcceptNodeCode {
   FILTER_ACCEPT   = 1,
   FILTER_REJECT   = 2,
   FILTER_SKIP     = 3
    } AcceptNodeCode;
```

CompareHowCode

Defines type of comparison.

Definition

```
typedef enum CompareHowCode {
   START_TO_START = 0,
   START_TO_END = 1,
   END_TO_END = 2,
   END_TO_START = 3 }
CompareHowCode;
```

DOMNodeType

Defines type of node.

Definition

```
typedef enum DOMNodeType {
   UNDEFINED_NODE = 0,
   ELEMENT_NODE = 1,
   ATTRIBUTE_NODE = 2,
   TEXT_NODE = 3,
   CDATA_SECTION_NODE = 4,
   ENTITY_REFERENCE_NODE = 5,
   ENTITY_NODE = 6,
   PROCESSING_INSTRUCTION_NODE = 7,
   COMMENT_NODE = 8,
   DOCUMENT_NODE = 9,
   DOCUMENT_TYPE_NODE = 10,
   DOCUMENT_FRAGMENT_NODE = 11,
   NOTATION_NODE = 12
} DOMNodeType;
```

DOMExceptionCode

Defines codes for DOM exception.

Definition

```
typedef enum DOMExceptionCode {
   UNDEFINED_ERR               = 0,
   INDEX_SIZE_ERR              = 1,
   DOMSTRING_SIZE_ERR          = 2,
   HIERARCHY_REQUEST_ERR       = 3,
   WRONG_DOCUMENT_ERR          = 4,
   INVALID_CHARACTER_ERR       = 5,
   NO_DATA_ALLOWED_ERR         = 6,
   NO_MODIFICATION_ALLOWED_ERR = 7,
   NOT_FOUND_ERR               = 8,
   NOT_SUPPORTED_ERR           = 9,
   INUSE_ATTRIBUTE_ERR         = 10,
   INVALID_STATE_ERR           = 11,
   SYNTAX_ERR                  = 12,
```

```
   INVALID_MODIFICATION_ERR    = 13,
   NAMESPACE_ERR               = 14,
   INVALID_ACCESS_ERR          = 15
} DOMExceptionCode;
```

WhatToShowCode

Defines codes for filtering.

Definition

```
typedef unsigned long WhatToShowCode;
   const unsigned long SHOW_ALL = 0xFFFFFFFF; c
   onst unsigned long SHOW_ELEMENT = 0x00000001;
   const unsigned long SHOW_ATTRIBUTE = 0x00000002;
   const unsigned long SHOW_TEXT = 0x00000004;
   const unsigned long SHOW_CDATA_SECTION = 0x00000008;
   const unsigned long SHOW_ENTITY_REFERENCE = 0x00000010;
   const unsigned long SHOW_ENTITY = 0x00000020;
   const unsigned long SHOW_PROCESSING_INSTRUCTION = 0x00000040;
   const unsigned long SHOW_COMMENT = 0x00000080;
   const unsigned long SHOW_DOCUMENT = 0x00000100;
   const unsigned long SHOW_DOCUMENT_TYPE = 0x00000200;
   const unsigned long SHOW_DOCUMENT_FRAGMENT = 0x00000400;
   const unsigned long SHOW_NOTATION = 0x00000800;
```

RangeExceptionCode

Codes for DOM Range exceptions.

Definition

```
typedef enum RangeExceptionCode {
   RANGE_UNDEFINED_ERR         = 0,
   BAD_BOUNDARYPOINTS_ERR      = 1,
   INVALID_NODE_TYPE_ERR       = 2
} RangeExceptionCode;
```

AttrRef Interface

Table 15–2 summarizes the methods of available through `AttrRef` interface.

Table 15–2 Summary of TreeWalker Methods; Dom Package

Function	Summary
AttrRef on page 15-7	Constructor.
getName on page 15-8	Return attribute's name.
getOwnerElement on page 15-8	Return attribute's owning element.
getSpecified on page 15-8	Return boolean indicating if an attribute was explicitly created.
getValue on page 15-9	Return attribute's value.
setValue on page 15-9	Set attribute's value.
~AttrRef on page 15-9	Public default destructor.

AttrRef

Class constructor.

Syntax	Description

`AttrRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given attribute node after a call to `createAttribute`.

`AttrRef(` `  const AttrRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(AttrRef) Node` reference object

getName

Returns the fully-qualified name of an attribute (in the data encoding) as a `NULL`-terminated string.

Syntax

```
oratext* getName() const;
```

Returns

`(oratext *)` name of attribute

getOwnerElement

Returns attribute's owning element

Syntax

```
Node* getOwnerElement();
```

Returns

`(Node*)` attribute's owning element node.

getSpecified

Returns the 'specified' value for an attribute. If the attribute was explicitly given a value in the original document, it is `TRUE`; otherwise, it is `FALSE`. If the node is not an attribute, returns `FALSE`. If the user sets attribute's value through DOM, its 'specified' value will be `TRUE`.

Syntax

```
boolean getSpecified() const;
```

Returns

(boolean) attribute's "specified" value

getValue

Returns the "value" (character data) of an attribute (in the data encoding) as NULL-terminated string. Character and general entities will have been replaced.

Syntax

```
oratext* getValue() const;
```

Returns

(oratext*) attribute's value

setValue

Sets the given attribute's value to data. The new value must be in the data encoding. It is not verified, converted, or checked. The attribute's 'specified' flag will be TRUE after setting a new value.

Syntax

```
void setValue(
   oratext* data)
throw (DOMException);
```

Parameter	Description
data	new value of attribute

~AttrRef

This is the default destructor.

Syntax

```
~AttrRef();
```

CDATASectionRef Interface

Table 15–3 summarizes the methods of available through CDATASectionRef interface.

Table 15–3 Summary of CDATASectionRef Methods; Dom Package

Function	Summary
CDATASectionRef on page 15-11	Constructor.
~CDATASectionRef on page 15-11	Public default destructor.

CDATASectionRef

Class constructor.

Syntax	Description

`CDATASectionRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given CDATA node after a call to `createCDATASection`.

`CDATASectionRef(` `  const CDATASectionRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(CDATASectionRef) Node` reference object

~CDATASectionRef

This is the default destructor.

Syntax

```
~CDATASectionRef();
```

CharacterDataRef Interface

Table 15–4 summarizes the methods of available through `CharacterDataRef` interface.

Table 15–4 Summary of CharacterDataRef Methods; Dom Package

Function	Summary
appendData on page 15-13	Append data to end of node's current data.
deleteData on page 15-14	Remove part of node's data.
freeString on page 15-14	Deallocate the string allocated by `substringData`.
getData on page 15-14	Return node's data.
getLength on page 15-15	Return length of node's data.
insertData on page 15-15	Insert string into node's current data.
replaceData on page 15-16	Replace part of node's data.
setData on page 15-16	Set node's data.
substringData on page 15-17	Get substring of node's data.

appendData

Append a string to the end of a `CharacterData` node's data. The appended data should be in the data encoding. It will not be verified, converted, or checked.

Syntax

```
void appendData(
   oratext* data)
throw (DOMException);
```

Parameter	Description
`data`	data to append

deleteData

Remove a range of characters from a `CharacterData` node's data. The offset is zero-based, so offset zero refers to the start of the data. Both offset and count are in characters, not bytes. If the sum of offset and count exceeds the data length then all characters from offset to the end of the data are deleted.

Syntax

```
void deleteData(
   ub4 offset,
   ub4 count)
throw (DOMException);
```

Parameter	Description
`offset`	character offset where deletion starts
`count`	number of characters to delete

freeString

Deallocates the string allocated by substringData(). It is Oracle's extension.

Syntax

```
void freeString(
   oratext* str);
```

Parameter	Description
`str`	string

getData

Returns the data for a `CharacterData` node (type text, comment or CDATA) in the data encoding.

Syntax

```
oratext* getData() const;
```

Returns

`(oratext*)` node's data

getLength

Returns the length of the data for a `CharacterData` node (type `Text`, `Comment` or `CDATA`) in characters (not bytes).

Syntax

```
ub4 getLength() const;
```

Returns

`(ub4)` length in characters (not bytes!) of node's data

insertData

Insert a string into a `CharacterData` node's data at the specified position. The inserted data must be in the data encoding. It will not be verified, converted, or checked. The offset is specified as characters, not bytes. The offset is zero-based, so inserting at offset zero prepends the data.

Syntax

```
void insertData(
   ub4 offset,
   oratext* data)
throw (DOMException);
```

Parameter	Description
`offset`	character offset where insertion starts
`data`	data to insert

replaceData

Replaces a range of characters in a `CharacterData` node's data with a new string. The offset is zero-based, so offset zero refers to the start of the data. The replacement data must be in the data encoding. It will not be verified, converted, or checked. The offset and count are both in characters, not bytes. If the sum of offset and count exceeds length, then all characters to the end of the data are replaced.

Syntax

```
void replaceData(
   ub4 offset,
   ub4 count,
   oratext* data)
throw (DOMException);
```

Parameter	Description
`offset`	offset
`count`	number of characters to replace
`data`	data

setData

Sets data for a `CharacterData` node (type text, comment or CDATA), replacing the old data. The new data is not verified, converted, or checked -- it should be in the data encoding.

Syntax

```
void setData(
   oratext* data)
throw (DOMException);
```

Parameter	Description
`data`	data

substringData

Returns a range of character data from a `CharacterData` node (type `Text`, `Comment` or `CDATA`). Since the data is in the data encoding, offset and count are in characters, not bytes. The beginning of the string is offset 0. If the sum of offset and count exceeds the length, then all characters to the end of the data are returned. The substring is permanently allocated in the context managed memory and should be explicitly deallocated by `freeString`

Syntax

```
oratext* substringData(
   ub4 offset,
   ub4 count)
throw (DOMException);
```

Parameter	Description
`offset`	offset
`count`	number of characters to extract

Returns

`(oratext *)` specified substring

CommentRef Interface

Table 15–5 summarizes the methods of available through `CommentRef` interface.

Table 15–5 Summary of CommentRef Methods; Dom Package

Function	Summary
CommentRef on page 15-18	Constructor.
~CommentRef on page 15-18	Public default destructor.

CommentRef

Class constructor.

Syntax	Description
`CommentRef( const NodeRef< Node>& node_ref, Node* nptr);`	Used to create references to a given comment node after a call to createComment.
`CommentRef( const CommentRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(CommentRef) Node` reference object

~CommentRef

This is the default destructor.

Syntax

```
~CommentRef();
```

DOMException Interface

Table 15–6 summarizes the methods of available through the `DOMException` interface.

Table 15–6 Summary of DOMException Methods; Dom Package

Function	Summary
getDOMCode on page 15-20	Get DOM exception code embedded in the exception.
getMesLang on page 15-20	Get current language encoding of error messages.
getMessage on page 15-21	Get Oracle XML error message.

getDOMCode

This is a virtual member function that defines a prototype for implementation defined member functions returning DOM exception codes, defined in `DOMExceptionCode`, of the exceptional situations during execution

Syntax

```
virtual DOMExceptionCode getDOMCode() const = 0;
```

Returns

`(DOMExceptionCode)` exception code

getMesLang

Virtual member function inherited from `XmlException`

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

DOMImplRef Interface

Table 15–7 summarizes the methods of available through `DOMImplRef` interface.

Table 15–7 Summary of DOMImplRef Methods; Dom Package

Function	Summary
DOMImplRef on page 15-22	Constructor.
createDocument on page 15-23	Create document reference.
createDocumentType on page 15-23	Create DTD reference.
getImplementation on page 15-24	Get `DOMImplementation` object associated with the document.
getNoMod on page 15-24	Get the 'no modification allowed' flag value.
hasFeature on page 15-25	Determine if DOM feature is implemented.
setContext on page 15-25	Set another context to a node.
~DOMImplRef on page 15-26	Public default destructor.

DOMImplRef

Class constructor.

Syntax	Description

`DOMImplRef(`
`  Context* ctx_ptr,`
`  DOMImplementation< Node>* impl_ptr);`
	Creates reference object to `DOMImplementation` object in a given context. Returns reference to the implementation object.

`DOMImplRef(` `  const DOMImplRef< Context, Node>& iref);`	It is needed to create other references to the implementation object; deletion flags are not copied.

`DOMImplRef(`
`  const DOMImplRef< Context, Node>& iref,`
`  Context* ctx_ptr);`
	It is needed to create references to the implementation object in a different context; deletion flags are not copied.

Parameter	Description
ctx_ptr	context pointer
impl_ptr	implementation

Returns

(DOMImplRef) reference to the implementation object

createDocument

Creates document reference

Syntax

```
DocumentRef< Node>* createDocument(
   oratext* namespaceURI,
   oratext* qualifiedName,
   DocumentTypeRef< Node>& doctype)
throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI of root element
qualifiedName	qualified name of root element
doctype	associated DTD node

Returns

(DocumentRef< Node>*) document reference

createDocumentType

Creates DTD reference

Syntax

```
DocumentTypeRef< Node>* createDocumentType(
   oratext* qualifiedName,
   oratext* publicId,
   oratext* systemId)
```

```
throw (DOMException);
```

Parameter	Description
qualifiedName	qualified name
publicId	external subset public Id
systemId	external subset system Id

Returns

(DocumentTypeRef< Node>*) DTD reference

getImplementation

Returns DOMImplementation object that was used to create this document. When the DOMImplementation object is destructed, all document trees associated with it are also destructed.

Syntax

```
DOMImplementation< Node>* getImplementation() const;
```

Returns

(DOMImplementation) DOMImplementation reference object

getNoMod

Get the 'no modification allowed' flag value. This is an Oracle extension.

Syntax

```
boolean getNoMod() const;
```

Returns

TRUE if flag's value is TRUE, FALSE otherwise

hasFeature

Determine if a DOM feature is implemented. Returns `TRUE` if the feature is implemented in the specified version, `FALSE` otherwise.

In level 1, the legal values for package are 'HTML' and 'XML' (case-insensitive), and the version is the string "1.0". If the version is not specified, supporting any version of the feature will cause the method to return `TRUE`.

DOM 1.0 features are "XML" and "HTML".

DOM 2.0 features are "Core", "XML", "HTML", "Views", "StyleSheets", "CSS", "CSS2", "Events", "UIEvents", "MouseEvents", "MutationEvents", "HTMLEvents", "Range", "Traversal"

Syntax

```
boolean hasFeature(
   oratext* feature,
   oratext* version);
```

Parameter	Description
`feature`	package name of feature
`version`	version of package

Returns

`(boolean)` is feature implemented?

setContext

It is needed to create node references in a different context

Syntax

```
void setContext(
   NodeRef< Node>& nref,
   Context* ctx_ptr);
```

Parameter	Description
nref	reference node
ctx_ptr	context pointer

~DOMImplRef

This is the default destructor. It cleans the reference to the implementation object. It is usually called by the environment. But it can be called by the user directly if necessary.

Syntax

```
~DOMImplRef();
```

DOMImplementation Interface

Table 15–8 summarizes the methods of available through `DOMImplementation` interface.

Table 15–8 Summary of DOMImplementation Methods; Dom Package

Function	Summary
DOMImplementation on page 15-27	Constructor.
getNoMod on page 15-27	Get the 'nomodificationallowed' flag value.
~DOMImplementation on page 15-28	Public default destructor.

DOMImplementation

Creates `DOMImplementation` object. Sets the 'no modifications allowed' flag to the parameter value.

Syntax

```
DOMImplementation(
   boolean no_mod);
```

Parameter	Description
`no_mod`	whether modifications are allowed (`FALSE`) or not allowed (`TRUE`)

Returns

`(DOMImplementation)` implementation object

getNoMod

Get the 'no modification allowed' flag value. This is an Oracle extension.

Syntax

```
boolean getNoMod() const;
```

Returns

TRUE if flag's value is TRUE, FALSE otherwise

~DOMImplementation

This is the default destructor. It removes all DOM trees associated with this object.

Syntax

```
~DOMImplementation();
```

DocumentFragmentRef Interface

Table 15–9 summarizes the methods of available through `DocumentFragmentRef` interface.

Table 15–9 Summary of DocumentFragmentRef Methods; Dom Package

Function	Summary
DocumentFragmentRef on page 15-29	Constructor.
~DocumentFragmentRef on page 15-29	Public default destructor.

DocumentFragmentRef

Class constructor.

Syntax	Description

`DocumentFragmentRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given fragment node after a call to `createDocumentFragment`.

`DocumentFragmentRef(` `  const DocumentFragmentRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(DocumentFragmentRef) Node` reference object

~DocumentFragmentRef

This is the default destructor.

Syntax

```
~DocumentFragmentRef() {}
```

DocumentRange Interface

Table 15–10 summarizes the methods of available through `DocumentRange` interface.

Table 15–10 Summary of DocumentRange Methods; Dom Package

Function	Summary
DocumentRange on page 15-31	Constructor.
createRange on page 15-31	Create new range object.
destroyRange on page 15-32	Destroys Range object.
~DocumentRange on page 15-32	Default destructor.

DocumentRange

Constructs the factory.

Syntax

```
DocumentRange();
```

Returns

`(DocumentRange)` new factory object

createRange

Create new range object.

Syntax

```
Range< Node>* createRange(
   DocumentRef< Node>& doc);
```

Parameter	Description
`doc`	reference to document node

Returns

`(Range*)` Pointer to new range

destroyRange

Destroys range object.

Syntax

```
void destroyRange(
   Range< Node>* range)
throw (DOMException);
```

Parameter	Description
`range`	range

~DocumentRange

Default destructor.

Syntax

```
~DocumentRange();
```

DocumentRef Interface

Table 15–11 summarizes the methods of available through DocumentRef interface.

Table 15–11 Summary of DocumentRef Methods; Dom Package

Function	Summary
DocumentRef on page 15-34	Constructor.
createAttribute on page 15-34	Create an attribute node.
createAttributeNS on page 15-35	Create an attribute node with namespace information.
createCDATASection on page 15-35	Create a CDATA node.
createComment on page 15-36	Create a comment node.
createDocumentFragment on page 15-36	Create a document fragment.
createElement on page 15-37	Create an element node.
createElementNS on page 15-37	Create an element node with names pace information.
createEntityReference on page 15-38	Create an entity reference node.
createProcessingInstruction on page 15-39	Create a `ProcessingInstruction` node
createTextNode on page 15-39	Create a text node.
getDoctype on page 15-40	Get DTD associated with the document.
getDocumentElement on page 15-40	Get top-level element of this document.
getElementById on page 15-41	Get an element given its ID.
getElementsByTagName on page 15-41	Get elements in the document by tag name.
getElementsByTagNameNS on page 15-42	Get elements in the document by tag name (namespace aware version).
getImplementation on page 15-43	Get `DOMImplementation` object associated with the document.
importNode on page 15-43	Import a node from another DOM.s
~DocumentRef on page 15-44	Public default destructor.

DocumentRef

This is a constructor.

Syntax	Description
`DocumentRef( const NodeRef< Node>& nref, Node* nptr);`	Default constructor.
`DocumentRef( const DocumentRef< Node>& nref);`	Copy constructor.

Parameter	Description
`nref`	reference to provide the context
`nptr`	referenced node

Returns

`(DocumentRef) Node` reference object

createAttribute

Creates an attribute node with the given name. This is the non-namespace aware function. The new attribute will have `NULL` namespace URI and prefix, and its local name will be the same as its name, even if the name specified is a qualified name. The new node is an orphan with no parent. The name is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createAttribute(
   oratext* name)
throw (DOMException);
```

Parameter	Description
`name`	name

Returns

(Node*) New attribute node

createAttributeNS

Creates an attribute node with the given namespace URI and qualified name. The new node is an orphan with no parent. The URI and qualified name are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createAttributeNS(
   oratext* namespaceURI,
   oratext* qualifiedName)
   throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI
qualifiedName	qualified name

Returns

(Node*) New attribute node

createCDATASection

Creates a CDATA section node with the given initial data (which should be in the data encoding). A CDATA section is considered verbatim and is never parsed; it will not be joined with adjacent text nodes by the normalize operation. The initial data may be NULL, if provided; it is not verified, converted, or checked. The name of a CDATA node is always "#cdata-section". The new node is an orphan with no parent. The CDATA is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createCDATASection(
   oratext* data)
throw (DOMException);
```

Parameter	Description
data	data for new node

Returns

(Node*) New CDATA node

createComment

Creates a comment node with the given initial data (which must be in the data encoding). The data may be NULL, if provided; it is not verified, converted, or checked. The name of the comment node is always "#comment". The new node is an orphan with no parent. The comment data is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createComment(
   oratext* data)
throw (DOMException);
```

Parameter	Description
data	data for new node

Returns

(Node*) New comment node

createDocumentFragment

Creates an empty Document Fragment node. A document fragment is treated specially when it is inserted into a DOM tree: the children of the fragment are inserted in order instead of the fragment node itself. After insertion, the fragment node will still exist, but have no children. The name of a fragment node is always "#document-fragment".

Syntax

```
Node* createDocumentFragment()
throw (DOMException);
```

Returns

`(Node*)` new document fragment node

createElement

Creates an element node with the given tag name (which should be in the data encoding). The new node is an orphan with no parent. The `tagname` is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Note that the tag name of an element is case sensitive. This is the non-namespace aware function: the new node will have `NULL` namespace URI and prefix, and its local name will be the same as its tag name, even if the tag name specified is a qualified name.

Syntax

```
Node* createElement(
   oratext* tagname)
throw (DOMException);
```

Parameter	Description
`tagname`	tag name

Returns

`(Node*)` New element node

createElementNS

Creates an element with the given namespace URI and qualified name. The new node is an orphan with no parent. The URI and qualified name are not copied, their pointers are just stored. The user is responsible for persistence and freeing of that data.

Note that element names are case sensitive, and the qualified name is required though the URI may be NULL. The qualified name will be split into prefix and local parts. The tagName will be the full qualified name.

Syntax

```
Node* createElementNS(
   oratext* namespaceURI,
   oratext* qualifiedName)
throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI
qualifiedName	qualified name

Returns

(Node*) New element node

createEntityReference

Creates an entity reference node; the name (which should be in the data encoding) is the name of the entity to be referenced. The named entity does not have to exist. The name is not verified, converted, or checked. The new node is an orphan with no parent. The entity reference name is not copied; its pointer is just stored. The user is responsible for persistence and freeing of that data.

Note that entity reference nodes are never generated by the parser; instead, entity references are expanded as encountered. On output, an entity reference node will turn into a "&name;" style reference.

Syntax

```
Node* createEntityReference(
   oratext* name)
throw (DOMException);
```

Parameter	Description
name	name

Returns

(Node*) New entity reference node

createProcessingInstruction

Creates a processing instruction node with the given target and data (which should be in the data encoding). The data may be NULL, but the target is required and cannot be changed. The target and data are not verified, converted, or checked. The name of the node is the same as the target. The new node is an orphan with no parent. The target and data are not copied; their pointers are just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createProcessingInstruction(
   oratext* target,
   oratext* data)
throw (DOMException);
```

Parameter	Description
target	target
data	data for new node

Returns

(Node*) New PI node

createTextNode

Creates a text node with the given initial data (which must be non-NULL and in the data encoding). The data may be NULL; if provided, it is not verified, converted, checked, or parsed (entities will not be expanded). The name of the node is always "#text". The new node is an orphan with no parent. The text data is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

Syntax

```
Node* createTextNode(
   oratext* data)
throw (DOMException);
```

Parameter	Description
data	data for new text node

Returns

(Node*) new text node

getDoctype

Returns the DTD node associated with this document. After this call, a DocumentTypeRef object needs to be created with an appropriate constructor in order to call its member functions. The DTD tree cannot be edited.

Syntax

```
Node* getDoctype() const;
```

Returns

(Node*) DTD node

getDocumentElement

Returns the root element (node) of the DOM tree. Each document has only one uppermost Element node, called the root element. If there is no root element, NULL is returned. This can happen when the document tree is being constructed.

Syntax

```
Node* getDocumentElement() const;
```

Returns

(Node*) Root element

getElementById

Returns the element node which has the given ID. Throws NOT_FOUND_ERR if no element is found. The given ID should be in the data encoding or it might not match.

Note that attributes named "ID" are not automatically of type ID; ID attributes (which can have any name) must be declared as type ID in the DTD or XML schema associated with the document.

Syntax

```
Node* getElementById(
   oratext* elementId);
```

Parameter	Description
elementId	element id

Returns

(Node*)Element node

getElementsByTagName

Returns a list of all elements in the document with a given tag name, in document order (the order in which they would be encountered in a preorder traversal of the tree). The list should be freed by the user when it is no longer needed. The list is not live, it is a snapshot. That is, if a new node which matched the tag name were added to the DOM after the list was returned, the list would not automatically be updated to include the node.

The special name "*" matches all tag names; a NULL name matches nothing. Note that tag names are case sensitive, and should be in the data encoding or a mismatch might occur.

This function is not namespace aware; the full tag names are compared. If two qualified names with two different prefixes both of which map to the same URI are compared, the comparison will fail.

Syntax

```
NodeList< Node>* getElementsByTagName(
   oratext* tagname) const;
```

Parameter	Description
tagname	tag name

Returns

(NodeList< Node>*) List of nodes

getElementsByTagNameNS

Returns a list of all elements in the document with a given namespace URI and local name, in document order (the order in which they would be encountered in a preorder traversal of the tree). The list should be freed by the user when it is no longer needed. The list is not live, it is a snapshot. That is, if a new node which matches the URI and local name were added to the DOM after the list was returned, the list would not automatically be updated to include the node.

The URI and local name should be in the data encoding. The special name "*" matches all local names; a NULL local name matches nothing. Namespace URIs must always match, however, no wildcard is allowed. Note that comparisons are case sensitive.

Syntax

```
NodeList< Node>* getElementsByTagNameNS(
   oratext* namespaceURI,
   oratext* localName);
```

Parameter	Description
namespaceURI	
localName	

Returns

(NodeList< Node>*) List of nodes

getImplementation

Returns DOMImplementation object that was used to create this document. When the DOMImplementation object is destructed, all document trees associated with it are also destructed.

Syntax

```
DOMImplementation< Node>* getImplementation() const;
```

Returns

`(DOMImplementation)` DOMImplementation reference object

importNode

Imports a node from one Document to another. The new node is an orphan and has no parent. The original node is not modified in any way or removed from its document; instead, a new node is created with copies of all the original node's qualified name, prefix, namespace URI, and local name.

The deep controls whether the children of the node are recursively imported. If `FALSE`, only the node itself is imported, and it will have no children. If `TRUE`, all descendents of the node will be imported as well, and an entire new subtree created. Elements will have only their specified attributes imported; non-specified (default) attributes are omitted. New default attributes (for the destination document) are then added. Document and `DocumentType` nodes cannot be imported.

Syntax

```
Node* importNode(
   NodeRef< Node>& importedNode,
   boolean deep) const
throw (DOMException);
```

Parameter	Description
`importedNode`	
`deep`	

Returns

`(Node*)` New imported node

~DocumentRef

This is the default destructor. It cleans the reference to the node. If the document node is marked for deletion, the destructor deletes the node and the tree under it. It is always deep deletion in the case of a document node. The destructor can be called by the environment or by the user directly.

Syntax

```
~DocumentRef();
```

DocumentTraversal Interface

Table 15–12 summarizes the methods of available through DocumentTraversal interface.

Table 15–12 Summary of DocumentTraversal Methods; Dom Package

Function	Summary
DocumentTraversal on page 15-45	Constructor.
createNodeIterator on page 15-45	Create new `NodeIterator` object.
createTreeWalker on page 15-46	Create new `TreeWalker` object.
destroyNodeIterator on page 15-46	Destroys `NodeIterator` object.
destroyTreeWalker on page 15-47	Destroys `TreeWalker` object.
~DocumentTraversal on page 15-47	Default destructor.

DocumentTraversal

Constructs the factory.

Syntax

```
DocumentTraversal();
```

Returns

`(DocumentTraversal)` new factory object

createNodeIterator

Create new iterator object.

Syntax

```
NodeIterator< Node>* createNodeIterator(
   NodeRef< Node>& root,
   WhatToShowCode whatToShow,
   boolean entityReferenceExpansion)
```

```
throw (DOMException);
```

Parameter	Description
root	root of subtree, for iteration
whatToShow	node types filter
entityReferenceExpansion	if TRUE, expand entity references

Returns

(NodeIterator*) Pointer to new iterator

createTreeWalker

Create new TreeWalker object.

Syntax

```
TreeWalker< Node>* createTreeWalker(
   NodeRef< Node>& root,
   WhatToShowCode whatToShow,
   boolean entityReferenceExpansion)
throw (DOMException);
```

Parameter	Description
root	root of subtree, for traversal
whatToShow	node types filter
entityReferenceExpansion	if TRUE, expand entity references

Returns

(TreeWalker*) Pointer to new tree walker

destroyNodeIterator

Destroys node iterator object.

Syntax

```
void destroyNodeIterator(
   NodeIterator< Node>* iter)
throw (DOMException);
```

Parameter	Description
iter	iterator

destroyTreeWalker

Destroys `TreeWalker` object.

Syntax

```
void destroyTreeWalker(
   TreeWalker< Node>* walker)
throw (DOMException);
```

Parameter	Description
walker	`TreeWalker`

~DocumentTraversal

Default destructor.

Syntax

```
~DocumentTraversal();
```

DocumentTypeRef Interface

Table 15–13 summarizes the methods of available through DocumentTypeRef interface.

Table 15–13 Summary of DocumentTypeRef Methods; Dom Package

Function	Summary
DocumentTypeRef on page 15-48	Constructor.
getEntities on page 15-49	Get DTD's entities.
getInternalSubset on page 15-49	Get DTD's internal subset.
getName on page 15-49	Get name of DTD.
getNotations on page 15-50	Get DTD's notations.
getPublicId on page 15-50	Get DTD's public ID.
getSystemId on page 15-50	Get DTD's system ID.
~DocumentTypeRef on page 15-51	Public default destructor.

DocumentTypeRef

This is a constructor.

Syntax	Description

`DocumentTypeRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Default constructor.

`DocumentTypeRef(` `  const DocumentTypeRef< Node>& node_ref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(DocumentTypeRef) Node` reference object

getEntities

Returns a named node map of general entities defined by the DTD.

Syntax

```
NamedNodeMap< Node>* getEntities() const;
```

Returns

`(NamedNodeMap< Node>*)` map containing entities

getInternalSubset

Returns the content model for an element. If there is no DTD, returns `NULL`.

Syntax

```
Node* getInternalSubset(
   oratext* name);
```

Parameter	Description
`name`	name of element

Returns

`(xmlnode*)` content model subtree

getName

Returns DTD's name (specified immediately after the `DOCTYPE` keyword)

Syntax

```
oratext* getName() const;
```

Returns

`(oratext*)` name of DTD

getNotations

Returns a named node map of notations declared by the DTD.

Syntax

```
NamedNodeMap< Node>* getNotations() const;
```

Returns

`(NamedNodeMap< Node>*)` map containing notations

getPublicId

Returns DTD's public identifier

Syntax

```
oratext* getPublicId() const;
```

Returns

`(oratext*)` DTD's public identifier

getSystemId

Returns DTD's system identifier

Syntax

```
oratext* getSystemId() const;
```

Returns

`(oratext*)` DTD's system identifier

~DocumentTypeRef

This is the default destructor.

Syntax

```
~DocumentTypeRef();
```

ElementRef Interface

Table 15–14 summarizes the methods of available through `ElementRef` interface.

Table 15–14 Summary of ElementRef Methods; Dom Package

Function	Summary
ElementRef on page 15-52	Constructor.
getAttribute on page 15-53	Get attribute's value given its name.
getAttributeNS on page 15-54	Get attribute's value given its URI and local name.
getAttributeNode on page 15-54	Get the attribute node given its name.
getElementsByTagName on page 15-55	Get elements with given tag name.
getTagName on page 15-55	Get element's tag name.
hasAttribute on page 15-55	Check if named attribute exists.
hasAttributeNS on page 15-56	Check if named attribute exists (namespace aware version).
removeAttribute on page 15-56	Remove attribute with specified name.
removeAttributeNS on page 15-57	Remove attribute with specified URI and local name.
removeAttributeNode on page 15-57	Remove attribute node
setAttribute on page 15-58	Set new attribute for this element and/or new value.
setAttributeNS on page 15-58	Set new attribute for the element and/or new value.
setAttributeNode on page 15-59	Set attribute node.
~ElementRef on page 15-59	Public default destructor.

ElementRef

Class constructor.

Syntax	Description
`ElementRef( const NodeRef< Node>& node_ref, Node* nptr);`	Used to create references to a given element node after a call to createElement.
`ElementRef( const ElementRef< Node>& node_ref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(ElementRef) Node` reference object

getAttribute

Returns the value of an element's attribute (specified by name). Note that an attribute may have the empty string as its value, but cannot be `NULL`.

Syntax

```
oratext* getAttribute(
   oratext* name) const;
```

Parameter	Description
`name`	name of attribute (data encoding)

Returns

`(oratext*)` named attribute's value (in data encoding)

getAttributeNS

Returns the value of an element's attribute (specified by URI and local name). Note that an attribute may have the empty string as its value, but cannot be NULL.

Syntax

```
oratext* getAttributeNS(
   oratext* namespaceURI,
   oratext* localName);
```

Parameter	Description
namespaceURI	namespace URI of attribute (data encoding)
localName	local name of attribute (data encoding)

Returns

(oratext *) named attribute's value (in data encoding)

getAttributeNode

Returns the attribute node given its name.

Syntax

```
Node* getAttributeNode(
   oratext* name) const;
```

Parameter	Description
name	name of attribute (data encoding)

Returns

(Node*) the attribute node

getElementsByTagName

Returns a list of all elements with a given tag name, in the order in which they would be encountered in a preorder traversal of the subtree. The tag name should be in the data encoding. The special name "*" matches all tag names; a NULL name matches nothing. Tag names are case sensitive. This function is not namespace aware; the full tag names are compared. The returned list should be freed by the user.

Syntax

```
NodeList< Node>* getElementsByTagName(
   oratext* name);
```

Parameter	Description
name	tag name to match (data encoding)

Returns

(NodeList< Node>*) the list of elements

getTagName

Returns the tag name of an element node which is supposed to have the same value as the node name from the node interface

Syntax

```
oratext* getTagName() const;
```

Returns

(oratext*) element's name [in data encoding]

hasAttribute

Determines if an element has a attribute with the given name

Syntax

```
boolean hasAttribute(
   oratext* name);
```

Parameter	Description
name	name of attribute (data encoding)

Returns

(boolean) TRUE if element has attribute with given name

hasAttributeNS

Determines if an element has a attribute with the given URI and local name

Syntax

```
boolean hasAttributeNS(
   oratext* namespaceURI,
   oratext* localName);
```

Parameter	Description
namespaceURI	namespace URI of attribute (data encoding)
localName	local name of attribute (data encoding)

Returns

(boolean) TRUE if element has such attribute

removeAttribute

Removes an attribute specified by name. The attribute is removed from the element's list of attributes, but the attribute node itself is not destroyed.

Syntax

```
void removeAttribute(
   oratext* name) throw (DOMException);
```

Parameter	Description
name	name of attribute (data encoding)

removeAttributeNS

Removes an attribute specified by URI and local name. The attribute is removed from the element's list of attributes, but the attribute node itself is not destroyed.

Syntax

```
void removeAttributeNS(
   oratext* namespaceURI,
   oratext* localName)
throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI of attribute (data encoding)
localName	local name of attribute (data encoding)

removeAttributeNode

Removes an attribute from an element. Returns a pointer to the removed attribute or NULL

Syntax

```
Node* removeAttributeNode(
   AttrRef< Node>& oldAttr)
throw (DOMException);
```

Parameter	Description
oldAttr	old attribute node

Returns

(Node*) the attribute node (old) or NULL

setAttribute

Creates a new attribute for an element with the given name and value (which should be in the data encoding). If the named attribute already exists, its value is simply replaced. The name and value are not verified, converted, or checked. The value is not parsed, so entity references will not be expanded.

Syntax

```
void setAttribute(
   oratext* name,
   oratext* value)
throw (DOMException);
```

Parameter	Description
name	names of attribute (data encoding)
value	value of attribute (data encoding)

setAttributeNS

Creates a new attribute for an element with the given URI, local name and value (which should be in the data encoding). If the named attribute already exists, its value is simply replaced. The name and value are not verified, converted, or checked. The value is not parsed, so entity references will not be expanded.

Syntax

```
void setAttributeNS(
   oratext* namespaceURI,
   oratext* qualifiedName,
   oratext* value)
throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI of attribute (data encoding)
qualifiedName	qualified name of attribute(data encoding)
value	value of attribute(data encoding)

setAttributeNode

Adds a new attribute to an element. If an attribute with the given name already exists, it is replaced and a pointer to the old attribute returned. If the attribute is new, it is added to the element's list and a pointer to the new attribute is returned.

Syntax

```
Node* setAttributeNode(
   AttrRef< Node>& newAttr)
throw (DOMException);
```

Parameter	Description
newAttr	new node

Returns

(Node*) the attribute node (old or new)

~ElementRef

This is the default destructor.

Syntax

```
~ElementRef();
```

EntityRef Interface

Table 15–15 summarizes the methods of available through `EntityRef` interface.

Table 15–15 Summary of EntityRef Methods; Dom Package

Function	Summary
EntityRef on page 15-60	Constructor.
getNotationName on page 15-61	Get entity's notation.
getPublicId on page 15-61	Get entity's public ID.
getSystemId on page 15-61	Get entity's system ID.
getType on page 15-62	Get entity's type.
~EntityRef on page 15-62	Public default destructor.

EntityRef

Class constructor.

Syntax	Description
`EntityRef( const NodeRef< Node>& node_ref, Node* nptr);`	Used to create references to a given entity node after a call to create `Entity`.
`EntityRef( const EntityRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(EntityRef) Node` reference object

getNotationName

For unparsed entities, returns the name of its notation (in the data encoding). For parsed entities and other node types, returns `NULL`.

Syntax

```
oratext* getNotationName() const;
```

Returns

`(oratext*)` entity's notation

getPublicId

Returns an entity's public identifier (in the data encoding).

Syntax

```
oratext* getPublicId() const;
```

Returns

`(oratext*)` entity's public identifier

getSystemId

Returns an entity's system identifier (in the data encoding).

Syntax

```
oratext* getSystemId() const;
```

Returns

`(oratext*)` entity's system identifier

getType

Returns a boolean for an entity describing whether it is general (TRUE) or parameter (FALSE).

Syntax

```
boolean getType() const;
```

Returns

(boolean) TRUE for general entity, FALSE for parameter entity

~EntityRef

This is the default destructor.

Syntax

```
~EntityRef();
```

EntityReferenceRef Interface

Table 15–16 summarizes the methods of available through `EntityReferenceRef` interface.

Table 15–16 Summary of EntityReferenceRef Methods; Dom Package

Function	Summary
EntityReferenceRef on page 15-63	Constructor.
~EntityReferenceRef on page 15-63	Public default destructor.

EntityReferenceRef

Class constructor.

Syntax	Description

`EntityReferenceRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given entity reference node after a call to create `EntityReference`.

`EntityReferenceRef(` `  const EntityReferenceRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(EntityReferenceRef) Node` reference object

~EntityReferenceRef

This is the default destructor.

Syntax

```
~EntityReferenceRef();
```

NamedNodeMapRef Interface

Table 15–17 summarizes the methods of available through `NamedNodeMapRef` interface.

Table 15–17 Summary of NamedNodeMapRef Methods; Dom Package

Function	Summary
NamedNodeMapRef on page 15-65	Constructor
getLength on page 15-66	Get map's length
getNamedItem on page 15-66	Get item given its name
getNamedItemNS on page 15-66	Get item given its namespace URI and local name.
item on page 15-67	Get item given its index.
removeNamedItem on page 15-67	Remove an item given its name.
removeNamedItemNS on page 15-68	Remove the item from the map.
setNamedItem on page 15-68	Add new item to the map.
setNamedItemNS on page 15-69	Set named item to the map.
~NamedNodeMapRef on page 15-69	Default destructor.

NamedNodeMapRef

Class constructor.

Syntax	Description

`NamedNodeMapRef(`
`  const NodeRef< Node>& node_ref,`
`  NamedNodeMap< Node>* mptr);`
	Used to create references to a given `NamedNodeMap` node.

`NamedNodeMapRef(` `  const NamedNodeMapRef< Node>& mref);`	Copy constructor.

Parameter	Description
node_ref	reference to provide the context
nptr	referenced node

Returns

(NamedNodeMapRef) Node reference object

getLength

Get the length of the map.

Syntax

```
ub4 getLength() const;
```

Returns

(ub4) map's length

getNamedItem

Get the name of the item, given its name.

Syntax

```
Node* getNamedItem( oratext* name) const;
```

Parameter	Description
name	name of item

Returns

(Node*) pointer to the item

getNamedItemNS

Get the name of the item, given its namespace URI and local name.

Syntax

```
Node* getNamedItemNS(
   oratext* namespaceURI,
   oratext* localName) const;
```

Parameter	Description
namespaceURI	namespace URI of item
localName	local name of item

Returns

(Node*) pointer to the item

item

Get item, given its index.

Syntax

```
Node* item(
   ub4 index) const;
```

Parameter	Description
index	index of item

Returns

(Node*) pointer to the item

removeNamedItem

Remove the item from the map, given its name.

Syntax

```
Node* removeNamedItem(
   oratext* name)
throw (DOMException);
```

Parameter	Description
name	name of item

Returns

(Node*) pointer to the removed item

removeNamedItemNS

Remove the item from the map, given its namespace URI and local name.

Syntax

```
Node* removeNamedItemNS(
   oratext* namespaceURI,
   oratext* localName)
throw (DOMException);
```

Parameter	Description
namespaceURI	namespace URI of item
localName	local name of item

Returns

(Node*) pointer to the removed item

setNamedItem

Add new item to the map.

Syntax

```
Node* setNamedItem(
   NodeRef< Node>& newItem)
throw (DOMException);
```

Parameter	Description
newItem	item set to the map

Returns

`(Node*)` pointer to new item

setNamedItemNS

Set named item, which is namespace aware, to the map.

Syntax

```
Node* setNamedItemNS(
   NodeRef< Node>& newItem)
throw (DOMException);
```

Parameter	Description
`newItem`	item set to the map

Returns

`(Node*)` pointer to the item

~NamedNodeMapRef

Default destructor.

Syntax

```
~NamedNodeMapRef();
```

NodeFilter Interface

Table 15–18 summarizes the methods of available through NodeFilter interface.

Table 15–18 Summary of NodeFilter Methods; Dom Package

Function	Summary
acceptNode on page 15-70	Execute it for a given node and use its return value.

acceptNode

This function is used as a test by NodeIterator and `TreeWalker`.

Syntax

```
template< typename Node> AcceptNodeCode AcceptNode(
   NodeRef< Node>& nref);
```

Parameter	Description
`nref`	reference to the node to be tested.

Returns

`(AcceptNodeCode)` result returned by the filter function

NodeIterator Interface

Table 15–19 summarizes the methods of available through NodeIterator interface.

Table 15–19 Summary of NodeIterator Methods; Dom Package

Function	Summary
adjustCtx on page 15-71	Attach this iterator to the another context.
detach on page 15-71	Invalidate the iterator.
nextNode on page 15-72	Go to the next node.
previousNode on page 15-72	Go to the previous node.

adjustCtx

Attaches this iterator to the context associated with a given node reference

Syntax

```
void adjustCtx(
   NodeRef< Node>& nref);
```

Parameter	Description
nref	reference node

detach

Invalidates the iterator.

Syntax

```
void detach();
```

nextNode

Go to the next node.

Syntax

```
Node* nextNode() throw (DOMException);
```

Returns

`(Node*)` pointer to the next node

previousNode

Go to the previous node.

Syntax

```
Node* previousNode() throw (DOMException);
```

Returns

`(Node*)` pointer to the previous node

NodeListRef Interface

Table 15–20 summarizes the methods of available through NodeListRef interface.

Table 15–20 Summary of NodeListRef Methods; Dom Package

Function	Summary
NodeListRef on page 15-73	Constructor.
getLength on page 15-74	Get list's length.
item on page 15-74	Get item given its index.
~NodeListRef on page 15-74	Default destructor.

NodeListRef

Class constructor.

Syntax	Description

`NodeListRef(`
`  const NodeRef< Node>& node_ref,`
`  NodeList< Node>* lptr);`
	Used to create references to a given `NodeList` node.

`NodeListRef(` `  const NodeListRef< Node>& lref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`lptr`	referenced list

Returns

`(NodeListRef) Node` reference object

getLength

Get the length of the list.

Syntax

```
ub4 getLength() const;
```

Returns

`(ub4)` list's length

item

Get the item, given its index.

Syntax

```
Node* item(
   ub4 index) const;
```

Parameter	Description
`index`	index of item

Returns

`(Node*)` pointer to the item

~NodeListRef

Destructs the object.

Syntax

```
~NodeListRef();
```

NodeRef Interface

Table 15–21 summarizes the methods of available through `NodeRef` interface.

Table 15–21 Summary of NodeRef Methods; Dom Package

Function	Summary
NodeRef on page 15-76	Constructor.
appendChild on page 15-77	Append new child to node's list of children.
cloneNode on page 15-77	Clone this node.
getAttributes on page 15-78	Get attributes of this node.
getChildNodes on page 15-78	Get children of this node.
getFirstChild on page 15-79	Get the first child node of this node.
getLastChild on page 15-79	Get the last child node of this node.
getLocalName on page 15-79	Get local name of this node.
getNamespaceURI on page 15-80	Get namespace URI of this node as a `NULL`-terminated string.
getNextSibling on page 15-80	Get the next sibling node of this node.
getNoMod on page 15-80	Tests if no modifications are allowed for this node.
getNodeName on page 15-81	Get node's name as `NULL`-terminated string.
getNodeType on page 15-81	Get DOMNodeType of the node.
getNodeValue on page 15-81	Get node's value as `NULL`-terminated string.
getOwnerDocument on page 15-82	Get the owner document of this node.
getParentNode on page 15-82	Get parent node of this node.
getPrefix on page 15-82	Get namespace prefix of this node.
getPreviousSibling on page 15-83	Get the previous sibling node of this node.
hasAttributes on page 15-83	Tests if this node has attributes.
hasChildNodes on page 15-83	Test if this node has children.
insertBefore on page 15-84	Insert new child into node's list of children.

Table 15–21 (Cont.) Summary of NodeRef Methods; Dom Package

Function	Summary
isSupported on page 15-84	Tests if specified feature is supported by the implementation.
markToDelete on page 15-85	Sets the mark to delete the referenced node.
normalize on page 15-85	Merge adjacent text nodes.
removeChild on page 15-85	Remove an existing child node.
replaceChild on page 15-86	Replace an existing child of a node.
resetNode on page 15-86	Reset NodeRef to reference another node.
setNodeValue on page 15-87	Set node's value as NULL-terminated string.
setPrefix on page 15-87	Set namespace prefix of this node.
~NodeRef on page 15-88	Public default destructor.

NodeRef

Class constructor.

Syntax	Description
`NodeRef( const NodeRef< Node>& nref, Node* nptr);`	Used to create references to a given node when at least one reference to this node or another node is already available. The node deletion flag is not copied and is set to FALSE.
`NodeRef( const NodeRef< Node>& nref);`	Copy constructor. Used to create additional references to the node when at least one reference is already available. The node deletion flag is not copied and is set to FALSE.

Parameter	Description
node_ref	reference to provide the context
nptr	referenced node

Returns

`(NodeRef) Node` reference object

appendChild

Appends the node to the end of this node's list of children and returns the new node. If `newChild` is a `DocumentFragment`, all of its children are appended in original order; the `DocumentFragment` node itself is not. If `newChild` is already in the DOM tree, it is first removed from its current position.

Syntax

```
Node* appendChild(
   NodeRef& newChild)
throw (DOMException);
```

Parameter	Description
`newChild`	reference node

Returns

`(Node*)` the node appended

cloneNode

Creates and returns a duplicate of this node. The duplicate node has no parent. Cloning an Element copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but it does not copy any text it contains unless it is a deep clone, since the text is contained in a child `Text` node. Cloning any other type of node simply returns a copy of the node. If deep is `TRUE`, all children of the node are recursively cloned, and the cloned node will have cloned children; a non-deep clone will have no children. If the cloned node is not inserted into another tree or fragment, it probably should be marked, through its reference, for deletion (by the user).

Syntax

```
Node* cloneNode(
   boolean deep);
```

Parameter	Description
deep	whether to clone the entire node hierarchy beneath the node (TRUE), or just the node itself (FALSE)

Returns

(Node*) duplicate (cloned) node

getAttributes

Returns NamedNodeMap of attributes of this node, or NULL if it has no attributes. Only element nodes can have attribute nodes. For other node kinds, NULL is always returned. In the current implementation, the node map of child nodes is live; all changes in the original node are reflected immediately. Because of this, side effects can be present for some DOM tree manipulation styles, in particular, in multithreaded environments.

Syntax

```
NamedNodeMap< Node>* getAttributes() const;
```

Returns

(NamedNodeMap*) NamedNodeMap of attributes

getChildNodes

Returns the list of child nodes, or NULL if this node has no children. Only Element, Document, DTD, and DocumentFragment nodes may have children; all other types will return NULL. In the current implementation, the list of child nodes is live; all changes in the original node are reflected immediately. Because of this, side effects can be present for some DOM tree manipulation styles, in particular, in multithreaded environments.

Syntax

```
NodeList< Node>* getChildNodes() const;
```

Returns

`(NodeList*)` the list of child nodes

getFirstChild

Returns the first child node, or `NULL`, if this node has no children

Syntax

```
Node* getFirstChild() const;
```

Returns

`(Node*)` the first child node, or `NULL`

getLastChild

Returns the last child node, or `NULL`, if this node has no children

Syntax

```
Node* getLastChild() const;
```

Returns

`(Node*)` the last child node, or `NULL`

getLocalName

Returns local name (local part of the qualified name) of this node (in the data encoding) as a `NULL`-terminated string. If this node's name is not fully qualified (has no prefix), then the local name is the same as the name.

Syntax

```
oratext* getLocalName() const;
```

Returns

`(oratext*)` local name of this node

getNamespaceURI

Returns the namespace URI of this node (in the data encoding) as a NULL-terminated string. If the node's name is not qualified (does not contain a namespace prefix), it will have the default namespace in effect when the node was created (which may be NULL).

Syntax

```
oratext* getNamespaceURI() const;
```

Returns

(oratext*) namespace URI of this node

getNextSibling

Returns the next sibling node, or NULL, if this node has no next sibling

Syntax

```
Node* getNextSibling() const;
```

Returns

(Node*) the next sibling node, or NULL

getNoMod

Tests if no modifications are allowed for this node and the DOM tree it belongs to. This member function is Oracle extension.

Syntax

```
boolean getNoMod() const;
```

Returns

(boolean) TRUE if no modifications are allowed

getNodeName

Returns the (fully-qualified) name of the node (in the data encoding) as a NULL-terminated string, for example "bar\0" or "foo:bar\0". Some node kinds have fixed names: "#text", "#cdata-section", "#comment", "#document", "#document-fragment". The name of a node cannot changed once it is created.

Syntax

```
oratext* getNodeName() const;
```

Returns

(oratext*) name of node in data encoding

getNodeType

Returns DOMNodeType of the node

Syntax

```
DOMNodeType getNodeType() const;
```

Returns

(DOMNodeType) of the node

getNodeValue

Returns the "value" (associated character data) for a node as a NULL-terminated string. Character and general entities will have been replaced. Only Attr, CDATA, Comment, ProcessingInstruction and Text nodes have values, all other node types have NULL value.

Syntax

```
oratext* getNodeValue() const;
```

Returns

`(oratext *)` value of node

getOwnerDocument

Returns the document node associated with this node. It is assumed that the document node type is derived from the node type. Each node may belong to only one document, or may not be associated with any document at all, such as immediately after it was created on user's request. The "owning" document [node] is returned, or the WRONG_DOCUMENT_ERR exception is thrown.

Syntax

```
Node* getOwnerDocument() const throw (DOMException);
```

Returns

`(Node*)` the owning document node

getParentNode

Returns the parent node, or `NULL`, if this node has no parent

Syntax

```
Node* getParentNode() const;
```

Returns

`(Node*)` the parent node, or `NULL`

getPrefix

Returns the namespace prefix of this node (in the data encoding) (as a `NULL`-terminated string). If this node's name is not fully qualified (has no prefix), `NULL` is returned.

Syntax

```
oratext* getPrefix() const;
```

Returns

(oratext*) namespace prefix of this node

getPreviousSibling

Returns the previous sibling node, or NULL, if this node has no previous siblings

Syntax

```
Node* getPreviousSibling() const;
```

Returns

(Node*) the previous sibling node, or NULL

hasAttributes

Returns TRUE if this node has attributes, if it is an element. Otherwise, it returns FALSE. Note that for nodes that are not elements, it always returns FALSE.

Syntax

```
boolean hasAttributes() const;
```

Returns

(boolean) TRUE is this node is an element and has attributes

hasChildNodes

Tests if this node has children. Only Element, Document, DTD, and DocumentFragment nodes may have children.

Syntax

```
boolean hasChildNodes() const;
```

Returns

(boolean) TRUE if this node has any children

insertBefore

Inserts the node `newChild` before the existing child node `refChild` in this node. `refChild` must be a child of this node. If `newChild` is a `DocumentFragment`, all of its children are inserted (in the same order) before `refChild`; the `DocumentFragment` node itself is not. If `newChild` is already in the DOM tree, it is first removed from its current position.

Syntax

```
Node* insertBefore(
   NodeRef& newChild,
   NodeRef& refChild)
throw (DOMException);
```

Parameter	Description
`newChild`	new node
`refChild`	reference node

Returns

`(Node*)` the node being inserted

isSupported

Tests if the feature, specified by the arguments, is supported by the DOM implementation of this node.

Syntax

```
boolean isSupported(
   oratext* feature,
   oratext* version) const;
```

Parameter	Description
`feature`	package name of feature
`version`	version of package

Returns

`(boolean) TRUE` is specified feature is supported

markToDelete

Sets the mark indicating that the referenced node should be deleted at the time when destructor of this reference is called. All other references to the node become invalid. This behavior is inherited by all other reference classes. This member function is Oracle extension.

Syntax

```
void markToDelete();
```

normalize

"Normalizes" the subtree rooted at an element, merges adjacent `Text` nodes children of elements. Note that adjacent `Text` nodes will never be created during a normal parse, only after manipulation of the document with DOM calls.

Syntax

```
void normalize();
```

removeChild

Removes the node from this node's list of children and returns it. The node is orphaned; its parent will be `NULL` after removal.

Syntax

```
Node* removeChild(
   NodeRef& oldChild)
throw (DOMException);
```

Parameter	Description
`oldChild`	old node

Returns

(Node*) node removed

replaceChild

Replaces the child node oldChild with the new node newChild in this node's children list, and returns oldChild (which is now orphaned, with a NULL parent). If newChild is a DocumentFragment, all of its children are inserted in place of oldChild; the DocumentFragment node itself is not. If newChild is already in the DOM tree, it is first removed from its current position.

Syntax

```
Node* replaceChild(
   NodeRef& newChild,
   NodeRef& oldChild)
throw (DOMException);
```

Parameter	Description
newChild	new node
oldChild	old node

Returns

(Node*) the node replaced

resetNode

This function resets NodeRef to reference Node given as an argument

Syntax

```
void resetNode(
   Node* nptr);
```

Parameter	Description
nptr	reference node

setNodeValue

Sets a node's value (character data) as a NULL-terminated string. Does not allow setting the value to NULL. Only Attr, CDATA, Comment, ProcessingInstruction, and Text nodes have values. Trying to set the value of another kind of node is a no-op. The new value must be in the data encoding! It is not verified, converted, or checked. The value is not copied, its pointer is just stored. The user is responsible for persistence and freeing of that data.

It throws the NO_MODIFICATION_ALLOWED_ERR exception, if no modifications are allowed, or UNDEFINED_ERR, with an appropriate Oracle XML error code (see xml.h), in the case of an implementation defined error.

Syntax

```
void setNodeValue(
   oratext* data)
throw (DOMException);
```

Parameter	Description
data	new value for node

setPrefix

Sets the namespace prefix of this node (as NULL-terminated string). Does not verify the prefix is defined! And does not verify that the prefix is in the current data encoding. Just causes a new qualified name to be formed from the new prefix and the old local name.

It throws the NO_MODIFICATION_ALLOWED_ERR exception, if no modifications are allowed. Or it throws NAMESPACE_ERR if the namespaceURI of this node is NULL, or if the specified prefix is "xml" and the namespaceURI of this node is different from "http://www.w3.org/XML/1998/namespace", or if this node is an attribute and the specified prefix is "xmlns" and the namespaceURI of this node is different from "http://www.w3.org/2000/xmlns/". Note that the INVALID_CHARACTER_ERR exception is never thrown since it is not checked how the prefix is formed

Syntax

```
void setPrefix(
   oratext* prefix)
throw (DOMException);
```

Parameter	Description
prefix	new namespace prefix

~NodeRef

This is the default destructor. It cleans the reference to the node and, if the node is marked for deletion, deletes the node. If the node was marked for deep deletion, the tree under the node is also deleted (deallocated). It is usually called by the environment. But it can be called by the user directly if necessary.

Syntax

```
~NodeRef();
```

NotationRef Interface

Table 15–22 summarizes the methods of available through `NotationRef` interface.

Table 15–22 Summary of NotationRef Methods; Dom Package

Function	Summary
NotationRef on page 15-89	Constructor.
getPublicId on page 15-90	Get public ID.
getSystemId on page 15-90	Get system ID.
~NotationRef on page 15-90	Public default destructor.

NotationRef

Class constructor.

Syntax	Description

`NotationRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given notation node after a call to create `Notation`.

`NotationRef(` `  const NotationRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(NotationRef) Node` reference object

getPublicId

Get public id.

Syntax

```
oratext* getPublicId() const;
```

Returns

`(oratext*)` public ID

getSystemId

Get system id.

Syntax

```
oratext* getSystemId() const;
```

Returns

`(oratext*)` system ID

~NotationRef

This is the default destructor.

Syntax

```
~NotationRef();
```

ProcessingInstructionRef Interface

Table 15–23 summarizes the methods of available through `ProcessingInstructionRef` interface.

Table 15–23 Summary of ProcessingInstructionRef Methods; Dom Package

Function	Summary
ProcessingInstructionRef on page 15-91	Constructor.
getData on page 15-92	Get processing instruction's data.
getTarget on page 15-92	Get processing instruction's target.
setData on page 15-92	Set processing instruction's data.
~ProcessingInstructionRef on page 15-93	Public default destructor.

ProcessingInstructionRef

Class constructor.

Syntax	Description

`ProcessingInstructionRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given `ProcessingInstruction` node after a call to create `ProcessingInstruction`.

`ProcessingInstructionRef(` `  const ProcessingInstructionRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(ProcessingInstructionRef) Node` reference object

getData

Returns the content (data) of a processing instruction (in the data encoding). The content is the part from the first non-whitespace character after the target until the ending "?>".

Syntax

```
oratext* getData() const;
```

Returns

`(oratext*)` processing instruction's data

getTarget

Returns a processing instruction's target string. The target is the first token following the markup that begins the `ProcessingInstruction`. All `ProcessingInstructions` must have a target, though the data part is optional.

Syntax

```
oratext* getTarget() const;
```

Returns

`(oratext*)` processing instruction's target

setData

Sets processing instruction's data (content), which must be in the data encoding. It is not permitted to set the data to `NULL`. The new data is not verified, converted, or checked.

Syntax

```
void setData(
   oratext* data)
```

```
throw (DOMException);
```

Parameter	Description
data	new data

~ProcessingInstructionRef

This is the default destructor.

Syntax

```
~ProcessingInstructionRef();
```

Range Interface

Table 15–24 summarizes the methods of available through Range interface.

Table 15–24 Summary of Range Methods; Dom Package

Function	Summary
CompareBoundaryPoints on page 15-95	
cloneContent on page 15-95	
cloneRange on page 15-95	
deleteContents on page 15-96	
detach on page 15-96	Invalidate the range.
extractContent on page 15-96	
getCollapsed on page 15-96	Check if the range is collapsed.
getCommonAncestorContainer on page 15-97	Get the deepest common ancestor node.
getEndContainer on page 15-97	Get end container node.
getEndOffset on page 15-97	Get offset of the end point.
getStartContainer on page 15-98	Get start container node.
getStartOffset on page 15-98	Get offset of the start point.
insertNode on page 15-98	
selectNodeContent on page 15-99	
selectNode on page 15-99	
setEnd on page 15-99	Set end point.
setEndAfter on page 15-100	
setEndBefore on page 15-100	
setStart on page 15-101	Set start point.
setStartAfter on page 15-101	
setStartBefore on page 15-101	
surroundContents on page 15-102	
toString on page 15-102	

CompareBoundaryPoints

Compares boundary points.

Syntax

```
CompareHowCode compareBoundaryPoints(
   unsigned short how,
   Range< Node>* sourceRange)
throw (DOMException);
```

Parameter	Description
how	how to compare
sourceRange	range of comparison

Returns

(CompareHowCode) result of comparison

cloneContent

Makes a clone of the node, including its children.

Syntax

```
Node* cloneContents() throw (DOMException);
```

Returns

(Node*) subtree cloned

cloneRange

Clones a range of nodes.

Syntax

```
Range< Node>* cloneRange();
```

Returns

`(Range< Node>*)` new cloned range

deleteContents

Deletes contents of the node.

Syntax

```
void deleteContents() throw (DOMException);
```

detach

Invalidates the range. It is not recommended to use this method since it leaves the object in invalid state. The preferable way is to call the destructor.

Syntax

```
void detach();
```

extractContent

Extract the node.

Syntax

```
Node* extractContents() throw (DOMException);
```

Returns

`(Node*)` subtree extracted

getCollapsed

Checks if the range is collapsed.

Syntax

```
boolean getCollapsed() const;
```

Returns

`(boolean) TRUE` if the range is collapsed, `FALSE` otherwise

getCommonAncestorContainer

Get the deepest common ancestor of the node.

Syntax

```
Node* getCommonAncestorContainer() const;
```

Returns

`(Node*)` common ancestor node

getEndContainer

Gets the container node.

Syntax

```
Node* getEndContainer() const;
```

Returns

`(Node*)` end container node

getEndOffset

Get offset of the end point.

Syntax

```
long getEndOffset() const;
```

Returns

`(long)` offset

getStartContainer

Get start container node.

Syntax

```
Node* getStartContainer() const;
```

Returns

`(Node*)` start container node

getStartOffset

Get offset of the start point.

Syntax

```
long getStartOffset() const;
```

Returns

`(long)` offset

insertNode

Inserts a node.

Syntax

```
void insertNode(
   NodeRef< Node>& newNode)
throw (RangeException, DOMException);
```

Parameter	Description
newNode	inserted node

selectNodeContent

Selects node content by its reference.

Syntax

```
void selectNodeContent(
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

selectNode

Selects a node.

Syntax

```
void selectNode(
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

setEnd

Sets an end point.

Syntax

```
void setEnd(
   NodeRef< Node>& refNode,
   long offset)
```

```
throw (RangeException, DOMException);
```

Parameter	Description
refNode	reference node
offset	offset

setEndAfter

Sets the end pointer after a specified node.

Syntax

```
void setEndAfter(
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

setEndBefore

Set the end before a specified node.

Syntax

```
void setEndBefore(
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

setStart

Sets start point.

Syntax

```
void setStart(
   NodeRef< Node>& refNode,
   long offset)
throw (RangeException, DOMException);
```

Parameter	Description
refNode	reference node
offset	offset

setStartAfter

Sets start pointer after a specified node.

Syntax

```
void setStartAfter(
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

setStartBefore

Sets start pointer before a specified node.

Syntax

```
void setStartBefore(
```

```
   NodeRef< Node>& refNode)
throw (RangeException);
```

Parameter	Description
refNode	reference node

surroundContents

Makes a node into a child of the specified node.

Syntax

```
void surroundContents(
   NodeRef< Node>& newParent)
throw (RangeException, DOMException);
```

Parameter	Description
newParent	parent node

toString

Converts an item into a string.

Syntax

```
oratext* toString();
```

Returns

`(oratext*)` string representation of the range

RangeException Interface

Table 15–25 summarizes the methods of available through `RangeException` interface.

Table 15–25 Summary of RangeException Methods; Dom Package

Function	Summary
getCode on page 15-103	Get Oracle XML error code embedded in the exception.
getMesLang on page 15-103	Get current language (encoding) of error messages.
getMessage on page 15-104	Get Oracle XML error message.
getRangeCode on page 15-104	Get Range exception code embedded in the exception.

getCode

Gets Oracle XML error code embedded in the exception. Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getMesLang

Gets the current language encoding of error messages. Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Get XML error message. Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

getRangeCode

This is a virtual member function that defines a prototype for implementation defined member functions returning Range exception codes, defined in `RangeExceptionCode`, of the exceptional situations during execution.

Syntax

```
virtual RangeExceptionCode getRangeCode() const = 0;
```

Returns

`(RangeExceptionCode)` exception code

TextRef Interface

Table 15–26 summarizes the methods of available through `TextRef` interface.

Table 15–26 Summary of NodeIterator Methods; Dom Package

Function	Summary
TextRef on page 15-105	Constructor.
splitText on page 15-106	Split text node into two.
~TextRef on page 15-106	Public default destructor.

TextRef

Class constructor.

Syntax	Description

`TextRef(`
`  const NodeRef< Node>& node_ref,`
`  Node* nptr);`
	Used to create references to a given text node after a call to `createtext`.

`TextRef(` `  const TextRef< Node>& nref);`	Copy constructor.

Parameter	Description
`node_ref`	reference to provide the context
`nptr`	referenced node

Returns

`(TextRef) Node` reference object

splitText

Splits a single text node into two text nodes; the original data is split between them. The offset is zero-based, and is in characters, not bytes. The original node is retained, its data is just truncated. A new text node is created which contains the remainder of the original data, and is inserted as the next sibling of the original. The new text node is returned.

Syntax

```
Node* splitText(
   ub4 offset)
throw (DOMException);
```

Parameter	Description
`offset`	character offset where to split text

Returns

`(Node*)` new node

~TextRef

This is the default destructor.

Syntax

```
~TextRef();
```

TreeWalker Interface

Table 15–27 summarizes the methods of available through `TreeWalker` interface.

Table 15–27 Summary of TreeWalker Methods; Dom Package

Function	Summary
adjustCtx on page 15-107	Attach this tree walker to another context.
firstChild on page 15-107	Get the first child of the current node.
lastChild on page 15-108	Get the last child of the current node.
nextNode on page 15-108	Get the next node.
nextSibling on page 15-108	Get the next sibling node.
parentNode on page 15-109	Get the parent of the current node.
previousNode on page 15-109	Get the previous node.
previousSibling on page 15-109	Get the previous sibling node.

adjustCtx

Attaches this tree walker to the context associated with a given node reference

Syntax

```
void adjustCtx(
   NodeRef< Node>& nref);
```

Parameter	Description
`nref`	reference to provide the context

firstChild

Get the first child of the current node.

Syntax

```
Node* firstChild();
```

Returns

`(Node*)` pointer to first child node

lastChild

Get the last child of the current node.

Syntax

```
Node* lastChild();
```

Returns

`(Node*)` pointer to last child node

nextNode

Get the next node.

Syntax

```
Node* nextNode();
```

Returns

`(Node*)` pointer to the next node

nextSibling

Get the next sibling node.

Syntax

```
Node* nextSibling();
```

Returns

`(Node*)` pointer to the next sibling node

parentNode

Get the parent of the current node.

Syntax

```
Node* parentNode();
```

Returns

`(Node*)` pointer to the parent node

previousNode

Get the previous node.

Syntax

```
Node* previousNode();
```

Returns

`(Node*)` pointer to previous node

previousSibling

Get the previous sibling node.

Syntax

```
Node* previousSibling();
```

Returns

`(Node*)` pointer to the previous sibling node

16

Package IO APIs for C++

This chapter contains these sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

IO Datatypes

Table 16–1 summarizes the datatypes of the IO package.

Table 16–1 Summary of Datatypes; IO Package

Datatype	Description
InputSourceType on page 16-2	Defines input source types.

InputSourceType

Defines input source types.

Definition

```
typedef enum InputSourceType {
   ISRC_URI = 1,
   ISRC_FILE = 2,
   ISRC_BUFFER = 3,
   ISRC_DOM = 4,
   ISRC_CSTREAM = 5 }
InputSourceType;
```

InputSource Interface

Table 16–2 summarizes the methods of available through the IO interface

Table 16–2 Summary of IO Package Interfaces

Function	Summary
getBaseURI on page 16-3	Get the base URI.
setBaseURI on page 16-3	Set the base URI.

getBaseURI

Gets the base URI. It is used by some input sources such as File and URI.

Syntax

```
oratext* getBaseURI() { return baseURI; }
```

Returns

`(oratext*)` base URI

getISrcType

Gets the input source type.

Syntax

```
InputSourceType getISrcType() const { return isrctype; }
```

Returns

`(InputSourceType)` input source type

setBaseURI

Sets the base URI. It is used by some input sources such as File and URI.

Syntax

```
void setBaseURI( oratext* base_URI)  baseURI = base_URI; }
```

17

Package OracleXml APIs for C++

OracleXml is the namespace for all XML C++ interaces. It includes class `XmlException`, the root for all exceptions in XML, and these namespaces:

- Ctx - namespace for TCtx related declarations, described in Chapter 14, "Package Ctx APIs for C++"
- Dom - namespace for DOM related declarations, described in Chapter 15, "Package Dom APIs for C++"
- IO - namespace for input and output source declarations, described in Chapter 16, "Package IO APIs for C++"
- Parser - namespace for parser and schema validator declarations, described in Chapter 18, "Package Parser APIs for C++"
- Tools - namespace for Tools::Factory related declarations, described in Chapter 19, "Package Tools APIs for C++"
- XPath - namespace for XPath related declarations, described in Chapter 20, "Package XPath APIs for C++"
- XPointer - namespace for XPointer related declarations, described in Chapter 21, "Package XPointer APIs for C++"
- Xsl - namespace for XSLT related declarations, described in Chapter 22, "Package Xsl APIs for C++"

This chapter contains this section:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XmlException Interface

`XMLException` is the root interface for all XML exceptions. Table 17–1 summarizes the methods of available through the `OracleXml` Package.

Table 17–1 Summary of OracleXml Package Interfaces

Function	Summary
getCode on page 17-2	Get Oracle XML error code embedded in the exception.
getMesLang on page 17-2	Get current language (encoding) of error messages.
getMessage on page 17-3	Get Oracle XML error message.

getCode

This is a virtual member function that defines a prototype for implementation defined functions returning Oracle XML error codes (like error codes defined in xml.h) of the exceptional situations during execution

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getMesLang

This is a virtual member function that defines a prototype for user defined functions returning current language (encoding) of error messages for the exceptional situations during execution

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext *)` Current language (encoding) of error messages

getMessage

This is a virtual member function that defines a prototype for implementation defined functions returning Oracle XML error messages of the exceptional situations during execution.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

18

Package Parser APIs for C++

Parser interfaces include: Parser exceptions, Validator, Parser, DOMParser, and SAXParser.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Parser Datatypes

Table 18–1 summarizes the datatypes of the `Parser` package.

Table 18–1 Summary of Datatypes; Parser Package

Datatype	Description
ParserExceptionCode on page 18-2	Parser implementation of exceptions.
DOMParserIdType on page 18-2	Defines parser identifiers.
SAXParserIdType on page 18-3	Defines type of node.
SchValidatorIdType on page 18-3	Defines validator identifiers.

ParserExceptionCode

Parser implementation of exceptions.

Definition

```
typedef enum ParserExceptionCode {
   PARSER_UNDEFINED_ERR = 0,
   PARSER_VALIDATION_ERR = 1,
   PARSER_VALIDATOR_ERR = 2,
   PARSER_BAD_ISOURCE_ERR = 3,
   PARSER_CONTEXT_ERR = 4,
   PARSER_PARAMETER_ERR = 5,
   PARSER_PARSE_ERR = 6,
   PARSER_SAXHANDLER_SET_ERR = 7,
   PARSER_VALIDATOR_SET_ERR = 8 }
ParserExceptionCode;
```

DOMParserIdType

Defines parser identifiers.

Definition

```
typedef enum DOMParserIdType {
      DOMParCXml         = 1
    } DOMParserIdType;
```

```
ypedef enum CompareHowCode {
   START_TO_START = 0,
   START_TO_END = 1,
   END_TO_END = 2,
   END_TO_START = 3 }
CompareHowCode;
```

SAXParserIdType

Defines parser identifiers.

Definition

```
typedef enum SAXParserIdType {
   SAXParCXml = 1 }
SAXParserIdType;
```

SchValidatorIdType

Defines validator identifiers.These identifiers are used as parameters to the XML tools factory when a particular validator object has to be created.

Definition

```
typedef enum SchValidatorIdType {
   SchValCXml        = 1
} SchValidatorIdType;
```

DOMParser Interface

Table 18–2 summarizes the methods of available through the `DOMParser` interface.

Table 18–2 Summary of DOMParser Methods; Parser Package

Function	Summary
getContext on page 18-4	Returns parser's XML context (allocation and encodings).
getParserId on page 18-4	Get parser id.
parse on page 18-5	Parse the document.
parseDTD on page 18-5	Parse DTD document.
parseSchVal on page 18-6	Parse and validate the document.
setValidator on page 18-6	Set the validator for this parser.

getContext

Each parser object is allocated and executed in a particular Oracle XML context. This member function returns a pointer to this context.

Syntax

```
virtual Context* getContext() const = 0;
```

Returns

`(Context*)` pointer to parser's context

getParserId

Syntax

```
virtual DOMParserIdType getParserId() const = 0;
```

Returns

`(DOMParserIdType)` Parser Id

parse

Parses the document and returns the tree root node

Syntax

```
virtual DocumentRef< Node>* parse(
   InputSource* isrc_ptr,
   boolean DTDvalidate = FALSE,
   DocumentTypeRef< Node>* dtd_ptr = NULL,
   boolean no_mod = FALSE,
   DOMImplementation< Node>* impl_ptr = NULL)
throw (ParserException) = 0;
```

Parameter	Description
`isrc_ptr`	input source
`DTDvalidate`	`TRUE` if validated by DTD
`dtd_ptr`	DTD reference
`no_mod`	`TRUE` if no modifications allowed
`impl_ptr`	optional `DomImplementation` pointer

Returns

`(DocumentRef)` document tree

parseDTD

Parse DTD document.

Syntax

```
virtual DocumentRef< Node>* parseDTD(
   InputSource* isrc_ptr,
   boolean no_mod = FALSE,
   DOMImplementation< Node>* impl_ptr = NULL)
throw (ParserException) = 0;
```

Parameter	Description
isrc_ptr	input source
no_mod	TRUE if no modifications allowed
impl_ptr	optional DomImplementation pointer

Returns

(DocumentRef) DTD document tree

parseSchVal

Parses and validates the document. Sets the validator if the corresponding parameter is not NULL.

Syntax

```
virtual DocumentRef< Node>* parseSchVal(
   InputSource* src_par,
   boolean no_mod = FALSE,
   DOMImplementation< Node>* impl_ptr = NULL,
   SchemaValidator< Node>* tor_ptr = NULL)
throw (ParserException) = 0;
```

Parameter	Description
isrc_ptr	input source
no_mod	TRUE if no modifications allowed
impl_ptr	optional DomImplementation pointer
tor_ptr	schema validator

Returns

(DocumentRef) document tree

setValidator

Sets the validator for all validations except when another one is given in parseSchVal

Syntax

```
virtual void setValidator(
SchemaValidator< Node>* tor_ptr) = 0;
```

Parameter	Description
tor_ptr	schema validator

GParser Interface

Table 18–3 summarizes the methods of available through the `GParser` interface.

Table 18–3 Summary of GParser Methods; Parser Package

Function	Summary
SetWarnDuplicateEntity on page 18-8	Specifies if multiple entity declarations result in a warning.
getBaseURI on page 18-9	Returns the base URI for the document.
getDiscardWhitespaces on page 18-9	Checks if whitespaces between elements are discarded.
getExpandCharRefs on page 18-9	Checks if character references are expanded.
getSchemaLocation on page 18-10	Get schema location for this document.
getStopOnWarning on page 18-10	Get if document processing stops on warnings.
getWarnDuplicateEntity on page 18-10	Get if multiple entity declarations cause a warning.
setBaseURI on page 18-11	Sets the base URI for the document.
setDiscardWhitespaces on page 18-11	Sets if formatting whitespaces should be discarded.
setExpandCharRefs on page 18-12	Get if character references are expanded.
setSchemaLocation on page 18-12	Set schema location for this document.
setStopOnWarning on page 18-12	Sets if document processing stops on warnings.

SetWarnDuplicateEntity

Specifies if entities that are declared more than once will cause warnings to be issued.

Syntax

```
void setWarnDuplicateEntity(
   boolean par_bool);
```

Parameter	Description
par_bool	TRUE if multiple entity declarations cause a warning

getBaseURI

Returns the base URI for the document. Usually only documents loaded from a URI will automatically have a base URI. Documents loaded from other sources (stdin, buffer, and so on) will not naturally have a base URI, but a base URI may have been set for them using setBaseURI, for the purposes of resolving relative URIs in inclusion.

Syntax

```
oratext* getBaseURI() const;
```

Returns

(oratext *) current document's base URI [or NULL]

getDiscardWhitespaces

Checks if formatting whitespaces between elements, such as newlines and indentation in input documents are discarded. By default, all input characters are preserved.

Syntax

```
boolean getDiscardWhitespaces() const;
```

Returns

(boolean) TRUE if whitespace between elements are discarded

getExpandCharRefs

Checks if character references are expanded in the DOM data. By default, character references are replaced by the character they represent. However, when a document

is saved those characters entities do not reappear. To ensure they remain through load and save, they should not be expanded.

Syntax

```
boolean getExpandCharRefs() const;
```

Returns

`(boolean) TRUE` if character references are expanded

getSchemaLocation

Gets schema location for this document. It is used to figure out the optimal layout when loading documents into a database.

Syntax

```
oratext* getSchemaLocation() const;
```

Returns

`(oratext*)` schema location

getStopOnWarning

When `TRUE` is returned, warnings are treated the same as errors and cause parsing, validation, and so on, to stop immediately. By default, warnings are issued but the processing continues.

Syntax

```
boolean getStopOnWarning() const;
```

Returns

`(boolean) TRUE` if document processing stops on warnings

getWarnDuplicateEntity

Get if entities which are declared more than once will cause warnings to be issued.

Syntax

```
boolean getWarnDuplicateEntity() const;
```

Returns

`(boolean) TRUE` if multiple entity declarations cause a warning

setBaseURI

Sets the base URI for the document. Usually only documents that were loaded from a URI will automatically have a base URI. Documents loaded from other sources (stdin, buffer, and so on) will not naturally have a base URI, but a base URI may have been set for them using setBaseURI, for the purposes of resolving relative URIs in inclusion.

Syntax

```
void setBaseURI( oratext* par);
```

Parameter	Description
`par`	base URI

setDiscardWhitespaces

Sets if formatting whitespaces between elements (newlines and indentation) in input documents are discarded. By default, ALL input characters are preserved.

Syntax

```
void setDiscardWhitespaces(
   boolean par_bool);
```

Parameter	Description
`par_bool`	`TRUE` if whitespaces should be discarded

setExpandCharRefs

Sets if character references should be expanded in the DOM data. Ordinarily, character references are replaced by the character they represent. However, when a document is saved those characters entities do not reappear. To ensure they remain through load and save is to not expand them.

Syntax

```
void setExpandCharRefs(
   boolean par_bool);
```

Parameter	Description
par_bool	TRUE if character references should be discarded

setSchemaLocation

Sets schema location for this document. It is used to figure out the optimal layout when loading documents into a database.

Syntax

```
 void setSchemaLocation(
   oratext* par);
```

Parameter	Description
par	schema location

setStopOnWarning

When TRUE is set, warnings are treated the same as errors and cause parsing, validation, and so on, to stop immediately. By default, warnings are issued but the processing continues.

Syntax

```
void setStopOnWarning(
   boolean par_bool);
```

Parameter	Description
par_bool	TRUE if document processing should stop on warnings

ParserException Interface

Table 18–4 summarizes the methods of available through the `ParserException` interface.

Table 18–4 Summary of ParserException Methods; Parser Package

Function	Summary
getCode on page 18-14	Get Oracle XML error code embedded in the exception.
getMesLang on page 18-14	Get current language (encoding) of error messages.
getMessage on page 18-15	Get Oracle XML error message.
getParserCode on page 18-15	Get parser exception code embedded in the exception.

getCode

Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getMesLang

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

getParserCode

This is a virtual member function that defines a prototype for implementation defined member functions returning parser and validator exception codes, defined in ParserExceptionCode, of the exceptional situations during execution.

Syntax

```
virtual ParserExceptionCode getParserCode() const = 0;
```

Returns

`(ParserExceptionCode)` exception code

SAXHandler Interface

Table 18–5 summarizes the methods of available through the `SAXHandler` interface.

Table 18–5 Summary of SAXHandler Methods; Parser Package

Function	Summary
CDATA on page 18-16	Receive notification of CDATA.
XMLDecl on page 18-17	Receive notification of an XML declaration.
attributeDecl on page 18-17	Receive notification of attribute's declaration.
characters on page 18-18	Receive notification of character data.
comment on page 18-18	Receive notification of a comment.
elementDecl on page 18-19	Receive notification of element's declaration.
endDocument on page 18-19	Receive notification of the end of the document.
endElement on page 18-19	Receive notification of element's end.
notationDecl on page 18-20	Receive notification of a notation declaration.
parsedEntityDecl on page 18-20	Receive notification of a parsed entity declaration.
processingInstruction on page 18-21	Receive notification of a processing instruction.
startDocument on page 18-21	Receive notification of the start of the document.
startElement on page 18-21	Receive notification of element's start.
startElementNS on page 18-22	Receive namespace aware notification of element's start.
unparsedEntityDecl on page 18-22	Receive notification of an unparsed entity declaration.
whitespace on page 18-23	Receive notification of whitespace characters.

CDATA

This event handles CDATA, as distinct from Text. The data will be in the data encoding, and the returned length is in characters, not bytes. This is an Oracle extension.

Syntax

```
virtual void CDATA(
   oratext* data,
   ub4 size) = 0;
```

Parameter	Description
data	pointer to CDATA
size	size of CDATA

XMLDecl

This event marks an XML declaration (XMLDecl). The startDocument event is always first; this event will be the second event. The encoding flag says whether an encoding was specified. For the standalone flag, -1 will be returned if it was not specified, otherwise 0 for FALSE, 1 for TRUE. This member function is an Oracle extension.

Syntax

```
virtual void XMLDecl(
   oratext* version,
   boolean is_encoding,
   sword standalone) = 0;
```

Parameter	Description
version	version string from XMLDecl
is_encoding	whether encoding was specified
standalone	value of standalone value flag

attributeDecl

This event marks an attribute declaration in the DTD. It is an Oracle extension; not in SAX standard

Syntax

```
virtual void attributeDecl(
   oratext* attr_name,
   oratext *name,
   oratext *content) = 0;
```

Parameter	Description
attr_name	
name	
content	body of attribute declaration

characters

This event marks character data.

Syntax

```
virtual void characters(
   oratext* ch,
   ub4 size) = 0;
```

Parameter	Description
ch	pointer to data
size	length of data

comment

This event marks a comment in the XML document. The comment's data will be in the data encoding. It is an Oracle extension, not in SAX standard.

Syntax

```
virtual void comment(
   oratext* data) = 0;
```

Parameter	Description
data	comment's data

elementDecl

This event marks an element declaration in the DTD. It is an Oracle extension; not in SAX standard.

Syntax

```
virtual void elementDecl(
   oratext *name,
   oratext *content) = 0;
```

Parameter	Description
name	element's name
content	element's content

endDocument

Receive notification of the end of the document.

Syntax

```
virtual void endDocument() = 0;
```

endElement

This event marks the end of an element. The name is the `tagName` of the element (which may be a qualified name for namespace-aware elements) and is in the data encoding.

Syntax

```
virtual void endElement( oratext* name) = 0;
```

notationDecl

The even marks the declaration of a notation in the DTD. The notation's name, public ID, and system ID will all be in the data encoding. Both IDs are optional and may be NULL.

Syntax

```
virtual void notationDecl(
   oratext* name,
   oratext* public_id,
   oratext* system_id) = 0;
```

Parameter	Description
name	notations's name
public_id	notation's public Id
sysem_id	notation's system Id

parsedEntityDecl

Marks a parsed entity declaration in the DTD. The parsed entity's name, public ID, system ID, and notation name will all be in the data encoding. This is an Oracle extension.

Syntax

```
virtual void parsedEntityDecl(
   oratext* name,
   oratext* value,
   oratext* public_id,
   oratext* system_id,
   boolean general) = 0;
```

Parameter	Description
name	entity's name
value	entity's value if internal
public_id	entity's public Id
sysem_id	entity's system Id
general	whether a general entity (FALSE if parameter entity)

processingInstruction

This event marks a processing instruction. The PI's target and data will be in the data encoding. There is always a target, but the data may be NULL.

Syntax

```
virtual void processingInstruction(
   oratext* target,
   oratext* data) = 0;
```

Parameter	Description
target	PI's target
data	PI's data

startDocument

Receive notification of the start of document.

Syntax

```
virtual void startDocument() = 0;
```

startElement

This event marks the start of an element.

Syntax

```
virtual void startElement(
   oratext* name,
   NodeListRef< Node>* attrs_ptr) = 0;
```

Parameter	Description
name	element's name
attrs_ptr	list of element's attributes

startElementNS

This event marks the start of an element. Note this is the new SAX 2 namespace-aware version. The element's qualified name, local name, and namespace URI will be in the data encoding, as are all the attribute parts.

Syntax

```
virtual void startElementNS(
   oratext* qname,
   oratext* local,
   oratext* ns_URI,
   NodeListRef< Node>* attrs_ptr) = 0;
```

Parameter	Description
qname	element's qualified name
local	element's namespace local name
ns_URI	element's namespace URI
attrs_ref	NodeList of element's attributes

unparsedEntityDecl

Marks an unparsed entity declaration in the DTD. The unparsed entity's name, public ID, system ID, and notation name will all be in the data encoding.

Syntax

```
virtual void unparsedEntityDecl(
   oratext* name,
   oratext* public_id,
   oratext* system_id,
   oratext* notation_name) = 0;
};
```

Parameter	Description
name	entity's name
public_id	entity's public Id
sysem_id	entity's system Id
notation_name	entity's notation name

whitespace

This event marks ignorable whitespace data such as newlines, and indentation between lines.

Syntax

```
virtual void whitespace(
   oratext* data,
   ub4 size) = 0;
```

Parameter	Description
data	pointer to data
size	length of data

SAXParser Interface

Table 18–6 summarizes the methods of available through the `SAXParser` interface.

Table 18–6 Summary of SAXParser Methods; Parser Package

Function	Summary
getContext on page 18-24	Returns parser's XML context (allocation and encodings).
getParserId on page 18-24	Returns parser Id.
parse on page 18-25	Parse the document.
parseDTD on page 18-25	Parse the DTD.
setSAXHandler on page 18-26	Set SAX handler.

getContext

Each parser object is allocated and executed in a particular Oracle XML context. This member function returns a pointer to this context.

Syntax

```
virtual Context* getContext() const = 0;
```

Returns

`(Context*)` pointer to parser's context

getParserId

Returns the parser id.

Syntax

```
virtual SAXParserIdType getParserId() const = 0;
```

Returns

`(SAXParserIdType)` Parser Id

parse

Parses a document.

Syntax

```
virtual void parse(
   InputSource* src_ptr,
   boolean DTDvalidate = FALSE,
   SAXHandlerRoot* hdlr_ptr = NULL)
throw (ParserException) = 0;
```

Parameter	Description
`src_ptr`	input source
`DTDValidate`	`TRUE` if validate with DTD
`hdlr_ptr`	SAX handler pointer

parseDTD

Parses a DTD.

Syntax

```
virtual void parseDTD(
   InputSource* src_ptr,
   SAXHandlerRoot* hdlr_ptr = NULL)
throw (ParserException) = 0;
```

Parameter	Description
`src_ptr`	input source
`hdlr_ptr`	SAX handler pointer

setSAXHandler

Sets SAX handler for all parser invocations except when another SAX handler is specified in the parser call.

Syntax

```
virtual void setSAXHandler(
   SAXHandlerRoot* hdlr_ptr) = 0;
```

Parameter	Description
hdlr_ptr	SAX handler pointer

SchemaValidator Interface

Table 18–7 summarizes the methods of available through the `SchemaValidator` interface.

Table 18–7 Summary of SchemaValidator Methods; Parser Package

Function	Summary
getSchemaList on page 18-27	Return the Schema list.
getValidatorId on page 18-27	Get validator identifier.
loadSchema on page 18-28	Load a schema document.
unloadSchema on page 18-28	Unload a schema document.

getSchemaList

Return only the size of loaded schema list documents if "list" is `NULL`. If "list" is not `NULL`, a list of URL pointers is returned in the user-provided pointer buffer. Note that its user's responsibility to provide a buffer with big enough size.

Syntax

```
virtual ub4 getSchemaList(
   oratext **list) const = 0;
```

Parameter	Description
`list`	address of a pointer buffer

Returns

`(ub4)` list size and list of loaded schemas (I/O parameter)

getValidatorId

Get the validator identifier corresponding to the implementation of this validator object.

Syntax

```
virtual SchValidatorIdType getValidatorId() const = 0;
```

Returns

`(SchValidatorIdType)` validator identifier

loadSchema

Load up a schema document to be used in the next validation session. Throws an exception in the case of an error.

Syntax

```
virtual void loadSchema(
   oratext* schema_URI)
throw (ParserException) = 0;
```

Parameter	Description
`schema_URI`	URL of a schema document; compiler encoding

unloadSchema

Unload a schema document and all its descendants (included or imported in a nested manner from the validator. All previously loaded schema documents will remain loaded until they are unloaded. To unload all loaded schema documents, set `schema_URI` to be `NULL`. Throws an exception in the case of an error.

Syntax

```
virtual void unloadSchema(
   oratext* schema_URI)
throw (ParserException) = 0;
```

Parameter	Description
`schema_URI`	URL of a schema document; compiler encoding

19

Package Tools APIs for C++

`Tools` packages contains types and interfaces related to the creation and instantiation of Oracle XML tools.

This chapter contains this section:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Tools Datatypes

Table 19–1 summarizes the datatypes of the `Tools` package.

Table 19–1 Summary of Datatypes; Tools Package

Datatype	Description
FactoryExceptionCode on page 19-2	Tool Factory exceptions.

FactoryExceptionCode

Tool Factory exceptions.

Definition

```
typedef enum FactoryExceptionCode {
   FACTORY_UNDEFINED_ERR = 0,
   FACTORY_OTHER_ERR = 1
} FactoryExceptionCode;
```

Factory Interface

Table 19–2 summarizes the methods of available through the `Factory` interface.

Table 19–2 Summary of Factory Methods; Tools Package

Function	Summary
Factory on page 19-3	Constructor.
createDOMParser on page 19-4	Create DOM Parser.
createSAXParser on page 19-4	Create SAX Parser.
createSchemaValidator on page 19-5	Create schema validator.
createXPathCompProcessor on page 19-5	Create extended XPath processor.
createXPathCompiler on page 19-6	Create XPath compiler.
createXPathProcessor on page 19-6	Create XPath processor.
createXPointerProcessor on page 19-7	Create XPointer processor.
createXslCompiler on page 19-7	Create Xsl compiler.
createXslExtendedTransformer on page 19-8	Create XSL extended transformer.
createXslTransformer on page 19-8	Create XSL transformer.
getContext on page 19-9	Get factory's context.
~Factory on page 19-9	Default destructor.

Factory

Class constructor.

Syntax	Description
`Factory()` `throw (FactoryException);`	Default constructor

`Factory(`
`  Context* ctx_ptr)`
`throw (FactoryException);`
	Creates factory object given a Context object.

Parameter	Description
ctx_ptr	pointer to a context object

Returns

(Factory) object

createDOMParser

Creates DOM parser.

Syntax

```
DOMParser< Context, Node>* createDOMParser (
   DOMParserIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	parser id type
ctx_ptr	pointer to a Context object

Returns

(DOMParser*) pointer to the parser object

createSAXParser

Creates SAX parser.

Syntax

```
SAXParser< Context>* createSAXParser (
   SAXParserIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	parser id type
ctx_ptr	pointer to a Context object

Returns

(SAXParser*) pointer to the parser object

createSchemaValidator

Creates schema validator.

Syntax

```
SchemaValidator< Node>* createSchemaValidator (
   SchValidatorIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	validator id type
ctx_ptr	pointer to a Context object

Returns

(SchemaValidator*) pointer to the validator object

createXPathCompProcessor

Creates extended XPath processor; takes XvmPrCXml value only.

Syntax

```
CompProcessor< Context, Node>* createXPathCompProcessor (
   XPathPrIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	processor id type
ctx_ptr	pointer to a Context object

Returns

(CompProcessor*) pointer to the processor object

createXPathCompiler

Creates XPath compiler.

Syntax

```
XPath::Compiler< Context, Node>* createXPathCompiler (
   XPathCompIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	compiler id type
ctx_ptr	pointer to a Context object

Returns

(XPathCompiler*) pointer to the compiler object

createXPathProcessor

Creates XPath processor.

Syntax

```
XPath::Processor< Context, Node>* createXPathProcessor (
   XPathPrIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	processor id type
ctx_ptr	pointer to a Context object

Returns

(Processor*) pointer to the processor object

createXPointerProcessor

Creates XPointer processor.

Syntax

```
XPointer::Processor< Context, Node>* createXPointerProcessor (
   XppPrIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	processor id type
ctx_ptr	pointer to a Context object

Returns

(Processor*) pointer to the processor object

createXslCompiler

Creates Xsl compiler.

Syntax

```
Xsl::Compiler< Context, Node>* createXslCompiler (
   XslCompIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	compiler id type
ctx_ptr	pointer to a Context object

Returns

(Compiler*) pointer to the compiler object

createXslExtendedTransformer

Creates XSL extended trnasformer; takes XvmTrCXml value only.

Syntax

```
CompTransformer< Context, Node>* createXslExtendedTransformer (
   XslTrIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	transformer id type
ctx_ptr	pointer to a Context object

Returns

(CompTrasformer*) pointer to the transformer object

createXslTransformer

Creates XSL trnasformer.

Syntax

```
Transformer< Context, Node>* createXslTransformer (
   XslTrIdType id_type,
   Context* ctx_ptr = NULL)
throw (FactoryException);
```

Parameter	Description
id_type	transformer id type
ctx_ptr	pointer to a Context object

Returns

(Trasformer*) pointer to the transformer object

getContext

Returns factory's context.

Syntax

```
Context* getContext() const;
```

Returns

(Context*) pointer to the context object

~Factory

Default destructor.

Syntax

```
~Factory();
```

FactoryException Interface

Table 19–3 summarizes the methods of available through the `FactoryException` interface.

Table 19–3 Summary of FactoryException Methods; Tools Package

Function	Summary
getCode on page 19-10	Get Oracle XML error code embedded in the exception.
getFactoryCode on page 19-10	Get FactoryException code embedded in the exception.
getMesLang on page 19-11	Get current language (encoding) of error messages.
getMessage on page 19-11	Get Oracle XML error message.

getCode

Gets Oracle XML error code embedded in the exception. Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getFactoryCode

This is a virtual member function that defines a prototype for implementation defined member functions returning exception codes specific to the Tools namespace, defined in FactoryExceptionCode, of the exceptional situations during execution

Syntax

```
virtual FactoryExceptionCode getFactoryCode() const = 0;
```

Returns

`(FactoryExceptionCode)` exception code

getMesLang

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

20

Package XPath APIs for C++

`XPath` package contains XPath processing related types and interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XPath Datatypes

Table 20–1 summarizes the datatypes of the `XPath` package.

Table 20–1 Summary of Datatypes; XPath Package

Datatype	Description
XPathCompIdType on page 20-2	Defines XPath compiler identifiers.
XPathObjType on page 20-2	Defines object types for XPath 1.0 based implementations.
XPathExceptionCode on page 20-3	XPath related exception codes.
XPathPrIdType on page 20-3	Defines XPath processor identifiers.

XPathCompIdType

Defines XPath compiler identifiers.

Definition

```
typedef enum XPathCompIdType {
   XvmXPathCompCXml = 1
} XPathCompIdType;
```

XPathObjType

Defines object types for XPath 1.0 based implementations.

Definition

```
typedef enum XPathObjType {
   XPOBJ_TYPE_UNKNOWN = 0,
   XPOBJ_TYPE_NDSET   = 1,
   XPOBJ_TYPE_BOOL    = 2,
   XPOBJ_TYPE_NUM     = 3,
   XPOBJ_TYPE_STR     = 4
} XPathObjType;
```

XPathExceptionCode

XPath related exception codes.

Definition

```
typedef enum XPathExceptionCode {
   XPATH_UNDEFINED_ERR = 0,
   XPATH_OTHER_ERR = 1
} XPathExceptionCode;
```

XPathPrIdType

Defines XPath processor identifiers.

Definition

```
typedef enum XPathPrIdType {
      XPathPrCXml       = 1,
      XvmPrCXml         = 2
    } XPathPrIdType;
```

CompProcessor Interface

Table 20–2 summarizes the methods of available through the `CompProcessor` interface.

Table 20–2 Summary of CompProcessor Methods; XPath Package

Function	Summary
getProcessorId on page 20-4	Get processor's Id.
process on page 20-4	Evaluate XPath expression against given document.
processWithBinXPath on page 20-5	Evaluate compiled XPath expression against given document.

getProcessorId

Get processor Id.

Syntax

```
virtual XPathPrIdType getProcessorId() const = 0;
```

Returns

`(XPathPrIdType)` Processor's Id

process

Inherited from Processor.

Syntax

```
virtual XPathObject< Node>* process (
   InputSource* isrc_ptr,
   oratext* xpath_exp)
throw (XPathException) = 0;
```

Parameter	Description
isrc_ptr	instance document to process
xpath_exp	XPATH expression

Returns

(XPathGenObject*) XPath object

processWithBinXPath

Evaluates compiled XPath expression against given document.

Syntax

```
virtual XPathObject< Node>* processWithBinXPath (
   InputSource* isrc_ptr,
   ub2* bin_xpath)
throw (XPathException) = 0;
```

Parameter	Description
isrc_ptr	instance document to process
bin_xpath	compiled XPATH expression

Returns

(XPathGenObject*) XPath object

Compiler Interface

Table 20–3 summarizes the methods of available through the `Compiler` interface.

Table 20–3 Summary of Compiler Methods; XPath Package

Function	Summary
compile on page 20-6	Compile XPath and return its compiled binary representation.
getCompilerId on page 20-6	Get the compiler's Id

compile

Compiles XPath and returns its compiled binary representation.

Syntax

```
virtual ub2* compile (
   oratext* xpath_exp)
throw (XPathException) = 0;
```

Parameter	Description
`xpath_exp`	XPATH expression

Returns

`(ub2)` XPath expression in compiled binary representation

getCompilerId

Get compiler's id.

Syntax

```
virtual XPathCompIdType getCompilerId() const = 0;
```

Returns

`(XPathCompIdType)` Compiler's Id

NodeSet Interface

Table 20–4 summarizes the methods of available through the `NodeSet` interface.

Table 20–4 Summary of NodeSet Methods; XPath Package

Function	Summary
getNode on page 20-8	Get node given its index.
getSize on page 20-8	Get `NodeSet` size.

getNode

Returns a reference to the node.

Syntax

```
NodeRef< Node>* getNode(
   ub4 idx) const;
```

Parameter	Description
`idx`	index of the node in the set

Returns

`(NodeRef)` reference to the node

getSize

The size of the node set.

Syntax

```
ub4 getSize() const;
```

Returns

`(ub4)` node set size

Processor Interface

Table 20–5 summarizes the methods of available through the `Processor` interface.

Table 20–5 Summary of Processor Methods; XPath Package

Function	Summary
getProcessorId on page 20-10	Get processor's Id.
process on page 20-10	Evaluate XPath expression against given document.

getProcessorId

Get processor Id.

Syntax

```
virtual XPathPrIdType getProcessorId() const = 0;
```

Returns

`(XPathPrIdType)` Processor's Id

process

Evaluates XPath expression against given document and returns result XPath object.

Syntax

```
virtual XPathObject< Node>* process (
   InputSource* isrc_ptr,
   oratext* xpath_exp)
throw (XPathException) = 0;
```

Parameter	Description
`isrc_ptr`	instance document to process

Parameter	Description
xpath_exp	XPath expression

Returns

(XPathGenObject*) XPath object

XPathException Interface

Table 20–6 summarizes the methods of available through the `XPathException` interface.

Table 20–6 Summary of XPathException Methods; XPath Package

Function	Summary
getCode on page 20-12	Get Oracle XML error code embedded in the exception.
getMesLang on page 20-12	Get current language (encoding) of error messages.
getMessage on page 20-13	Get Oracle XML error message.
getXPathCode on page 20-13	Get XPath exception code embedded in the exception.

getCode

Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getMesLang

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

getXPathCode

This is a virtual member function that defines a prototype for implementation defined member functions returning XPath processor and compiler exception codes, defined in XPathExceptionCode, of the exceptional situations during execution.

Syntax

```
virtual XPathExceptionCode getXPathCode() const = 0;
```

Returns

`(XPathExceptionCode)` exception code

XPathObject Interface

Table 20–7 summarizes the methods of available through the `XPathObject` interface.

Table 20–7 Summary of XPathObject Methods; XPath Package

Function	Summary
XPathObject on page 20-14	Copy constructor.
getNodeSet on page 20-14	Get the node set.
getObjBoolean on page 20-15	Get boolean from object.
getObjNumber on page 20-15	Get number from object.
getObjString on page 20-15	Get string from object.
getObjType on page 20-15	Get type from object.

XPathObject

Copy constructor.

Syntax

```
XPathObject(
   XPathObject< Node>& src);
```

Parameter	Description
`src`	reference to the object to be copied

Returns

`(XPathObject)` new object

getNodeSet

Get the node set.

Syntax

```
NodeSet< Node>* getNodeSet() const;
```

getObjBoolean

Get the boolean from the object.

Syntax

```
boolean getObjBoolean() const;
```

getObjNumber

Get the number from the object.

Syntax

```
double getObjNumber() const;
```

getObjString

Get the string from the object.

Syntax

```
oratext* getObjString() const;
```

getObjType

Get the type from the object.

Syntax

```
XPathObjType getObjType() const;
```

21

Package XPointer APIs for C++

`XPointer` package contains XPointer processing related types and interfaces.

This chapter contains the following sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

XPointer Datatypes

Table 21–1 summarizes the datatypes of the `XPointer` package.

Table 21–1 Summary of Datatypes; XPointer Package

Datatype	Description
XppExceptionCode on page 21-2	Defines XPath compiler identifiers.
XppPrIdType on page 21-2	Defines XPointer processor identifiers.
XppLocType on page 21-2	Defines location types for XPointer.

XppExceptionCode

XPointer related exception codes.

Definition

```
typedef enum XPathCompIdType {
   XvmXPathCompCXml = 1
} XPathCompIdType;
```

XppPrIdType

Defines XPointer processor identifiers.

Definition

```
typedef enum XppPrIdType {
   XPtrPrCXml         = 1
} XppPrIdType;
```

XppLocType

Defines location types for XPointer.

Definition

```
typedef enum XppLocType {
```

```
  XPPLOC_TYPE_UNKNOWN = 0,
  XPPLOC_TYPE_NODE    = 1,
  XPPLOC_TYPE_POINT   = 2,
  XPPLOC_TYPE_RANGE   = 3,
  XPPLOC_TYPE_BOOL    = 4,
  XPPLOC_TYPE_NUM     = 5,
  XPPLOC_TYPE_STR     = 6
} XppLocType;
```

Processor Interface

Table 21–2 summarizes the methods of available through the `Processor` interface.

Table 21–2 Summary of Processor Methods; XPointer Package

Function	Summary
getProcessorId on page 21-4	Get processor's Id.
process on page 21-4	Evaluate XPointer expression against given document.

getProcessorId

Get Processor Id.

Syntax

```
virtual XppPrIdType getProcessorId() const = 0;
```

Returns

`(XppPrIdType)` Processor's Id

process

Evaluates XPointer expression against given document and returns result XPointer location set object.

Syntax

```
virtual XppLocSet< Node>* process (
   InputSource* isrc_ptr,
   oratext* xpp_exp)
throw (XppException) = 0;
```

Parameter	Description
`isrc_ptr`	instance document to process

Parameter	Description
xpp_exp	XPointer expression

Returns

(XppLocSet*) XPath object

XppException Interface

Table 21–3 summarizes the methods of available through the `XPPException` interface.

Table 21–3 Summary of XppException Methods; Package XPointer

Function	Summary
getCode on page 21-6	Get Oracle XML error code embedded in the exception.
getMesLang on page 21-6	Get current language (encoding) of error messages.
getMesLang on page 21-6	Get Oracle XML error message.
getXppCode on page 21-7	Get XPointer exception code embedded in the exception.

getCode

Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (0 on success)

getMesLang

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

getXppCode

This is a virtual member function that defines a prototype for implementation defined member functions returning XPointer processor and compiler exception codes, defined in XppExceptionCode, of the exceptional situations during execution.

Syntax

```
virtual XppExceptionCode getXppCode() const = 0;
```

Returns

`(XppExceptionCode)` exception code

XppLocation Interface

Table 21–4 summarizes the methods of available through the `XppLocation` interface.

Table 21–4 Summary of XppLocation Methods; XPointer Package

Function	Summary
getLocType on page 21-8	Get the location type.
getNode on page 21-8	Get the node.
getRange on page 21-8	Get range.

getLocType

Get the location type.

Syntax

```
XppLocType getLocType() const;
```

getNode

Get the node.

Syntax

```
Node* getNode() const;
```

getRange

Get range.

Syntax

```
Range< Node>* getRange() const;
```

XppLocSet Interface

Table 21–5 summarizes the methods of available through the `XppLocSet` interface.

Table 21–5 Summary of XppLocSet Methods; XPointer Package

Function	Summary
getItem on page 21-9	Get item given its index.
getSize on page 21-9	Get location set size.

getItem

Returns a reference to the item.

Syntax

```
XppLocation< Node>* getItem(
  ub4 index) const;
```

Parameter	Description
`index`	index of an item

Returns

`(XppLocation*)` reference to the item

getSize

The size of the node set.

Syntax

```
ub4 getSize() const;
```

Returns

`(ub4)` node set size

22

Package Xsl APIs for C++

`Xsl` package contains XSLT related types and interfaces.

This chapter contains these sections:

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML DB Developer's Guide*

Xsl Datatypes

Table 22–1 summarizes the datatypes of the `Xsl` package.

Table 22–1 Summary of Datatypes; Xsl Package

Datatype	Description
XslCompIdType on page 22-2	Defines XSL compiler identifiers.
XslExceptionCode on page 22-2	Defines XSLT related exceptions.
XslTrIdType on page 22-2	Defines XSL transformer identifiers.

XslCompIdType

Defines XSL compiler identifiers.

Definition

```
typedef enum XslCompIdType {
   XvmCompCXml = 1
} XslCompIdType;
```

XslExceptionCode

Defines XSLT related exceptions.

Definition

```
typedef typedef enum XslExceptionCode {
   XSL_UNDEFINED_ERR = 0,
   XSL_OTHER_ERR = 1
} XslExceptionCode;
```

XslTrIdType

Defines XSL transformer identifiers.

Definition

```
typedef enum XslTrIdType {
   XslTrCXml          = 1,
   XvmTrCXml          = 2
} XslTrIdType;
```

Compiler Interface

Table 22–2 summarizes the methods of available through the `Compiler` interface.

Table 22–2 Summary of Compiler Methods; Xsl Package

Function	Summary
compile on page 22-4	Compile Xsl and return its compiled binary representation.
getCompilerId on page 22-4	Get compiler's Id.
getLength on page 22-5	Get length of compiled XSL document.

compile

Compiles Xsl and returns its compiled binary representation.

Syntax

```
virtual ub2* compile(
   InputSource* isrc_ptr)
throw (XslException) = 0;
```

Parameter	Description
`isrc_ptr`	Xsl document

Returns

`(InputSource)` Xsl document in compiled binary representation

getCompilerId

Get the compiler Id.

Syntax

```
virtual XslCompIdType getCompilerId() const = 0;
```

Returns

`(XslCompIdType)` Compiler's Id

getLength

Returns length of compiled XSL document

Syntax

```
virtual ub4 getLength(
   ub2* binxsl_ptr)
throw (XslException) = 0;
```

Parameter	Description
`binxsl_ptr`	compiled Xsl document

Returns

`(ub4)` length of the document

CompTransformer Interface

Table 22–3 summarizes the methods of available through the `CompTransformer` interface.

Table 22–3 Summary of CompTransformer Methods; Xsl Package

Function	Summary
getTransformerId	Get transformer's Id.
setBinXsl	Set compiled Xsl.
setSAXHandler	Set SAX handler.
setXSL	Set XSLT document for this transformer.
transform	Transform the document.

getTransformerId

Get transformer's id.

Syntax

```
virtual XslTrIdType getTransformerId() const = 0;
```

Returns

`(XslTrIdType)` Transformer's Id

setBinXsl

Sets compiled Xsl.

Syntax

```
virtual void setBinXsl (
   ub2* binxsl_ptr)
throw (XslException) = 0;
```

Parameter	Description
binxsl_ptr	compiled Xsl document

setSAXHandler

Inherited from Transformer.

Syntax

```
virtual void setSAXHandler(
   SAXHandlerRoot* hdlr_ptr) = 0;
```

Parameter	Description
hdlr_ptr	SAX handler pointer

setXSL

Set XSLT document for this transformer. Should be called before the transform member function is called. It is inherited from Transform.

Syntax

```
virtual void setXSL (
   InputSource* isrc_ptr)
throw (XslException) = 0;
```

Parameter	Description
isrc_ptr	instance document to process

transform

Transforms the document. Throws an exception if an XSLT document is not set by a previous call to setXSL. Inherited from Transform.

Syntax	Description
`virtual NodeRef< Node>* transform( nputSource* isrc_ptr) throw (XslException) = 0;`	Transform the document and return DOM.
`virtual void transform( InputSource* isrc_ptr, SAXHandlerRoot* hdlr_ptr) throw (XslException) = 0;`	Transform the document and return SAX events.

Parameter	Description
`isrc_ptr`	instance document to process
`hdlr_ptr`	SAX handler pointer

Returns

`(DocumentRef)` document tree of new document

Transformer Interface

Table 22–4 summarizes the methods of available through the `Transformer` interface.

Table 22–4 Summary of Transformer Methods; Xsl Package

Function	Summary
getTransformerId on page 22-9	Get transformer's Id.
setSAXHandler on page 22-9	Set SAX handler.
setXSL on page 22-10	Set XSLT document for this transformer.
transform on page 22-10	Transform the document and return SAX events.

getTransformerId

Gets transformer's id.

Syntax

```
virtual XslTrIdType getTransformerId() const = 0;
```

Returns

`(XslTrIdType)` Transformer's Id

setSAXHandler

Set SAX handler.

Syntax

```
virtual void setSAXHandler(
   SAXHandlerRoot* hdlr_ptr) = 0;
```

Parameter	Description
hdlr_ptr	SAX handler pointer

setXSL

Set XSLT document for this transformer. Should be called before the transform member function is called.

Syntax

```
virtual void setXSL (
   InputSource* isrc_ptr)
throw (XslException) = 0;
```

Parameter	Description
isrc_ptr	instance document to process

transform

Transforms the document. Throws an exception if an XSLT document is not set by a previous call to setXSL.

Syntax	Description

`virtual NodeRef< Node>* transform(`
`  nputSource* isrc_ptr)`
`throw (XslException) = 0;`
	Transform the document and return DOM.

`virtual void transform(`
`  InputSource* isrc_ptr,`
`  SAXHandlerRoot* hdlr_ptr)`
`throw (XslException) = 0;`
	Transform the document and return SAX events.

Parameter	Description
isrc_ptr	instance document to process
hdlr_ptr	SAX handler pointer

Returns

(DocumentRef) document tree of new document

XSLException Interface

Table 22–5 summarizes the methods of available through the `XSLException` interface.

Table 22–5 Summary of XSLException Methods; Xsl Package

Function	Summary
getCode on page 22-12	Get Oracle XML error code embedded in the exception.
getMesLang on page 22-12	Get current language (encoding) of error messages.
getMessage on page 22-13	Get Oracle XML error message.
getXslCode on page 22-13	Defines a prototype for implementation.

getCode

Gets Oracle XML error code embedded in the exception. Virtual member function inherited from `XmlException`.

Syntax

```
virtual unsigned getCode() const = 0;
```

Returns

`(unsigned)` numeric error code (`0` on success)

getMesLang

Virtual member function inherited from `XmlException`.

Syntax

```
virtual oratext* getMesLang() const = 0;
```

Returns

`(oratext*)` Current language (encoding) of error messages

getMessage

Virtual member function inherited from `XmlException`

Syntax

```
virtual oratext* getMessage() const = 0;
```

Returns

`(oratext *)` Error message

getXslCode

This is a virtual member function that defines a prototype for implementation defined member functions returning XSL transformer and compiler exception codes, defined in `XslExceptionCode`, of the exceptional situations during execution.

Syntax

```
virtual XslExceptionCode getXslCode() const = 0;
```

Returns

`(XslExceptionCode)` exception code