# Oracle® Database

# XML C++ API Reference

20c
F14765-01
February 2020

Oracle Database XML C++ API Reference, 20c

F14765-01



Primary Author: Jayashree Sharma

Contributing Authors: Tulika Das

Contributors: Roza Leyderman

# Contents

## 3 Package DOM APIs for C++

## 6 Package SOAP APIs for C++

## 7 Package Tools APIs for C++



## 8 Package XPath APIs for C++

## 9 Package XPointer APIs for C++

## 10 Package Xsl APIs for C++



## Index

# List of Tables

# Preface

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

## 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 the C programming language.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

**Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## Related Documents

For more information, see the following documents in the Oracle Database the 12*c* Release 2 (12.2) documentation set:

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

Additional information may be found at `http://www.oracle.com/technetwork/database-features/xdk/overview/index.html`

# Conventions

The following text conventions are used in this document:

| Convention | Meaning |
| --- | --- |
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |

1

# Package OracleXml APIs for C++

`OracleXml` is the namespace for all XML C++ interfaces implemented by Oracle. It includes class `XmlException`, the root for all exceptions in XML, and the following namespaces:

- Ctx - namespace for TCtx related declarations, described in Package Ctx APIs for C++
- Dom - namespace for DOM related declarations, described in Package Dom APIs for C++
- IO - namespace for input and output source declarations, described in Package IO APIs for C++
- Parser - namespace for parser and schema validator declarations, described in Package Parser APIs for C++
- SOAP - namespace for the Simple Object Access Protocol declarations, described in Package SOAP APIs for C++
- Tools - namespace for Tools::Factory related declarations, described in Package Tools APIs for C++
- XPath - namespace for XPath related declarations, described in Package XPath APIs for C++
- XPointer - namespace for XPointer related declarations, described in Package XPointer APIs for C++
- Xsl - namespace for XSLT related declarations, described in Package Xsl APIs for C++

> **Note:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## XmlException Interface

`XMLException` is the root interface for all XML exceptions. Table 1-1 summarizes the methods available through the `OracleXml` Package.

**Table 1-1 Summary of OracleXml Package Interfaces**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | 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

# 2
# Package Ctx APIs for C++

The XML context-related types and interfaces are represented in the `Ctx` namespace. These include `Ctx`datatypes, `MemAllocator`methods, and `TCtx` methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## Ctx Datatypes

Table 2-1 summarizes the datatypes of the `Ctx` package.

**Table 2-1 Summary of Datatypes; Ctx Package**

| Datatype | Description |
|---|---|
| encoding | A single supported encoding. |
| encodings | 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 2-2 summarizes the methods available through the `MemAllocator` interface.

**Table 2-2 Summary of MemAllocator Methods; Ctx Package**

| Function | Summary |
|---|---|
| alloc() | Allocates memory of given size. |
| dealloc() | Deallocate memory pointed to by the argument. |
| ~MemAllocator() | 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 2-3 summarizes the methods available through the TCtx interface.

**Table 2-3 Summary of TCtx Methods; Ctx Package**

| Function | Summary |
|---|---|
| TCtx() | Class constructor. |
| getEncoding() | Get data encoding in use by XML context. |
| getErrHandler() | Get Error Handler provided by the user. |
| getMemAllocator() | Get memory allocator. |
| isSimple() | Get a flag that indicates if data encoding is simple. |
| isUnicode() | Get a flag indicating if data encoding is Unicode. |
| ~TCtx() | 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(`<br>`  oratext* name,`<br>`  ErrorHandler* errh = NULL,`<br>`  MemAllocator* memalloc = NULL,`<br>`  encodings* encs = NULL)`<br>`throw (XMLException)` | This constructor creates the context object and initializes it with parameter values provided by the user. |
| `TCtx(`<br>`  oratext* name,`<br>`  up4 inpblksize,`<br>`  ErrorIfs* errh = NULL,`<br>`  MemAllocator* memalloc = NULL,`<br>`  encodings* encs = NULL)`<br>`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();
```

# 3
# Package DOM APIs for C++

The various interfaces in the DOM package represent DOM level 2 Core interfaces specified by W3C: http://www.w3.org/TR/DOM-Level-2-Core/.

## About DOM Interfaces

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 3-1 summarizes the datatypes of the Dom package.

**Table 3-1 Summary of Datatypes; Dom Package**

| Datatype | Description |
|---|---|
| AcceptNodeCodes | Defines values returned by node filters. |
| CompareHowCode | Defines type of comparison. |
| DOMNodeType | Defines type of node. |

**Table 3-1 (Cont.) Summary of Datatypes; Dom Package**

| Datatype | Description |
|---|---|
| DOMExceptionCode | Defines codes for DOM exception. |
| WhatToShowCode | Defines codes for filtering. |
| RangeExceptionCode | 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 3-2 summarizes the methods available through `AttrRef` interface.

**Table 3-2 Summary of AttrRef Methods; Dom Package**

| Function | Summary |
|---|---|
| AttrRef() | Constructor. |
| getName() | Return attribute's name. |
| getOwnerElement() | Return attribute's owning element. |
| getSpecified() | Return boolean indicating if an attribute was explicitly created. |
| getValue() | Return attribute's value. |
| setValue() | Set attribute's value. |
| ~AttrRef() | Public default destructor. |

## AttrRef()

Class constructor.

| Syntax | Description |
|---|---|
| `AttrRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given attribute node after a call to `createAttribute`. |
| `AttrRef(`<br>`  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 3-3 summarizes the methods available through CDATASectionRef interface.

**Table 3-3 Summary of CDATASectionRef Methods; Dom Package**

| Function | Summary |
|---|---|
| CDATASectionRef() | Constructor. |
| ~CDATASectionRef() | Public default destructor. |

## CDATASectionRef()

Class constructor.

| Syntax | Description |
|---|---|
| `CDATASectionRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given CDATA node after a call to `createCDATASection`. |
| `CDATASectionRef(`<br>`  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 3-4 summarizes the methods available through `CharacterDataRef` interface.

**Table 3-4 Summary of CharacterDataRef Methods; Dom Package**

| Function | Summary |
|---|---|
| appendData() | Append data to end of node's current data. |
| deleteData() | Remove part of node's data. |
| freeString() | Deallocate the string allocated by `substringData`. |
| getData() | Return node's data. |
| getLength() | Return length of node's data. |
| insertData() | Insert string into node's current data. |
| replaceData() | Replace part of node's data. |
| setData() | Set node's data. |
| substringData() | 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 3-5 summarizes the methods available through `CommentRef` interface.

**Table 3-5 Summary of CommentRef Methods; Dom Package**

| Function | Summary |
|---|---|
| CommentRef() | Constructor. |
| ~CommentRef() | 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 3-6 summarizes the methods available through the `DOMException` interface.

**Table 3-6 Summary of DOMException Methods; Dom Package**

| Function | Summary |
|---|---|
| getDOMCode() | Get DOM exception code embedded in the exception. |
| getMesLang() | Get current language encoding of error messages. |
| getMessage() | 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 3-7 summarizes the methods available through `DOMImplRef` interface.

**Table 3-7 Summary of DOMImplRef Methods; Dom Package**

| Function | Summary |
|---|---|
| DOMImplRef() | Constructor. |
| createDocument() | Create document reference. |
| createDocumentType() | Create DTD reference. |
| formDocument() | Forms document reference given a document pointer. |
| getImplementation() | Get `DOMImplementation` object associated with the document. |
| getNoMod() | Get the 'no modification allowed' flag value. |
| hasFeature() | Determine if DOM feature is implemented. |
| setContext() | Set another context to a node. |
| ~DOMImplRef() | 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

# formDocument()

Forms a document reference given a document pointer.

### Syntax

```
DocumentRef< Node>* formDocument( Node* node);
```

| Parameter | Description |
|---|---|
| `node` | pointer to the document node |

### Returns

(DocumentRef< Node>*) pointer to the document 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 3-8 summarizes the methods available through DOMImplementation interface.

**Table 3-8 Summary of DOMImplementation Methods; Dom Package**

| Function | Summary |
|---|---|
| DOMImplementation() | Constructor. |
| getNoMod() | Get the 'nomodificationallowed' flag value. |
| ~DOMImplementation() | 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 3-9 summarizes the methods available through `DocumentFragmentRef` interface.

**Table 3-9 Summary of DocumentFragmentRef Methods; Dom Package**

| Function | Summary |
|---|---|
| DocumentFragmentRef() | Constructor. |
| ~DocumentFragmentRef() | Public default destructor. |

## DocumentFragmentRef()

Class constructor.

| Syntax | Description |
|---|---|
| `DocumentFragmentRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given fragment node after a call to `createDocumentFragment`. |
| `DocumentFragmentRef(`<br>`  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 3-10 summarizes the methods available through `DocumentRange` interface.

**Table 3-10 Summary of DocumentRange Methods; Dom Package**

| Function | Summary |
|---|---|
| DocumentRange() | Constructor. |
| createRange() | Create new range object. |
| destroyRange() | Destroys Range object. |
| ~DocumentRange() | 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 3-11 summarizes the methods available through DocumentRef interface.

**Table 3-11 Summary of DocumentRef Methods; Dom Package**

| Function | Summary |
|---|---|
| DocumentRef() | Constructor. |
| createAttribute() | Create an attribute node. |
| createAttributeNS() | Create an attribute node with namespace information. |
| createCDATASection() | Create a CDATA node. |
| createComment() | Create a comment node. |
| createDocumentFragment() | Create a document fragment. |
| createElement() | Create an element node. |
| createElementNS() | Create an element node with names pace information. |
| createEntityReference() | Create an entity reference node. |
| createProcessingInstruction() | Create a `ProcessingInstruction` node |
| createTextNode() | Create a text node. |
| getDoctype() | Get DTD associated with the document. |
| getDocumentElement() | Get top-level element of this document. |

**Table 3-11 (Cont.) Summary of DocumentRef Methods; Dom Package**

| Function | Summary |
|---|---|
| getElementById() | Get an element given its ID. |
| getElementsByTagName() | Get elements in the document by tag name. |
| getElementsByTagNameNS() | Get elements in the document by tag name (namespace aware version). |
| getImplementation() | Get `DOMImplementation` object associated with the document. |
| importNode() | Import a node from another DOM.s |
| ~DocumentRef() | Public default destructor. |

# DocumentRef()

This is a constructor.

| Syntax | Description |
|---|---|
| `DocumentRef(`<br>`  const NodeRef< Node>& nref,`<br>`  Node* nptr);` | Default constructor. |
| `DocumentRef(`<br>`  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` | namespace URI to match |
| `localName` | local name to match |

**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` parameter 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` | node to import |
| `deep` | indicator for recursively importing the subtree |

### 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 3-12 summarizes the methods available through DocumentTraversal interface.

**Table 3-12 Summary of DocumentTraversal Methods; Dom Package**

| Function | Summary |
|---|---|
| DocumentTraversal() | Constructor. |
| createNodeIterator() | Create new `NodeIterator` object. |
| createTreeWalker() | Create new `TreeWalker` object. |
| destroyNodeIterator() | Destroys `NodeIterator` object. |
| destroyTreeWalker() | Destroys `TreeWalker` object. |
| ~DocumentTraversal() | 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 3-13 summarizes the methods available through DocumentTypeRef interface.

**Table 3-13 Summary of DocumentTypeRef Methods; Dom Package**

| Function | Summary |
|---|---|
| DocumentTypeRef() | Constructor. |
| getEntities() | Get DTD's entities. |
| getInternalSubset() | Get DTD's internal subset. |
| getName() | Get name of DTD. |
| getNotations() | Get DTD's notations. |
| getPublicId() | Get DTD's public ID. |
| getSystemId() | Get DTD's system ID. |
| ~DocumentTypeRef() | 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 3-14 summarizes the methods available through `ElementRef` interface.

**Table 3-14 Summary of ElementRef Methods; Dom Package**

| Function | Summary |
|---|---|
| ElementRef() | Constructor. |
| getAttribute() | Get attribute's value given its name. |
| getAttributeNS() | Get attribute's value given its URI and local name. |
| getAttributeNode() | Get the attribute node given its name. |
| getElementsByTagName() | Get elements with given tag name. |
| getTagName() | Get element's tag name. |
| hasAttribute() | Check if named attribute exists. |
| hasAttributeNS() | Check if named attribute exists (namespace aware version). |
| removeAttribute() | Remove attribute with specified name. |
| removeAttributeNS() | Remove attribute with specified URI and local name. |
| removeAttributeNode() | Remove attribute node |
| setAttribute() | Set new attribute for this element and/or new value. |
| setAttributeNS() | Set new attribute for the element and/or new value. |
| setAttributeNode() | Set attribute node. |
| ~ElementRef() | Public default destructor. |

## ElementRef()

Class constructor.

| Syntax | Description |
|---|---|
| `ElementRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given element node after a call to createElement. |
| `ElementRef(`<br>`  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 3-15 summarizes the methods available through `EntityRef` interface.

**Table 3-15 Summary of EntityRef Methods; Dom Package**

| Function | Summary |
|---|---|
| EntityRef()EntityRef() | Constructor. |
| getNotationName()getNotationName() | Get entity's notation. |
| getPublicId()getPublicId() | Get entity's public ID. |
| getSystemId()getSystemId() | Get entity's system ID. |
| getType()getType() | Get entity's type. |
| ~EntityRef()~EntityRef() | Public default destructor. |

## EntityRef()

Class constructor.

| Syntax | Description |
|---|---|
| `EntityRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given entity node after a call to create `Entity`. |
| `EntityRef(`<br>`  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 3-16 summarizes the methods available through `EntityReferenceRef` interface.

**Table 3-16 Summary of EntityReferenceRef Methods; Dom Package**

| Function | Summary |
|---|---|
| EntityReferenceRef() | Constructor. |
| ~EntityReferenceRef() | 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 3-17 summarizes the methods available through `NamedNodeMapRef` interface.

**Table 3-17 Summary of NamedNodeMapRef Methods; Dom Package**

| Function | Summary |
|---|---|
| NamedNodeMapRef() | Constructor |
| getLength() | Get map's length |
| getNamedItem() | Get item given its name |

**Table 3-17 (Cont.) Summary of NamedNodeMapRef Methods; Dom Package**

| Function | Summary |
|---|---|
| getNamedItemNS() | Get item given its namespace URI and local name. |
| item() | Get item given its index. |
| removeNamedItem() | Remove an item given its name. |
| removeNamedItemNS() | Remove the item from the map. |
| setNamedItem() | Add new item to the map. |
| setNamedItemNS() | Set named item to the map. |
| ~NamedNodeMapRef() | Default destructor. |

# NamedNodeMapRef()

Class constructor.

| Syntax | Description |
|---|---|
| `NamedNodeMapRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  NamedNodeMap< Node>* mptr);` | Used to create references to a given `NamedNodeMap` node. |
| `NamedNodeMapRef(`<br>`  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 3-18 summarizes the methods available through NodeFilter interface.

**Table 3-18 Summary of NodeFilter Methods; Dom Package**

| Function | Summary |
|---|---|
| acceptNode()acceptNode() | 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 3-19 summarizes the methods available through NodeIterator interface.

**Table 3-19 Summary of NodeIterator Methods; Dom Package**

| Function | Summary |
|---|---|
| adjustCtx() | Attach this iterator to the another context. |
| detach() | Invalidate the iterator. |
| nextNode() | Go to the next node. |
| previousNode() | 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 3-20 summarizes the methods available through NodeListRef interface.

**Table 3-20 Summary of NodeListRef Methods; Dom Package**

| Function | Summary |
|---|---|
| NodeListRef() | Constructor. |
| getLength() | Get list's length. |
| item() | Get item given its index. |
| ~NodeListRef() | Default destructor. |

## NodeListRef()

Class constructor.

| Syntax | Description |
|---|---|
| `NodeListRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  NodeList< Node>* lptr);` | Used to create references to a given `NodeList` node. |
| `NodeListRef(`<br>`  const NodeListRef< Node>& lref);` | Copy constructor. |

| Parameter | Description |
|---|---|
| `node_ref` | reference to provide the context |

| Parameter | Description |
|---|---|
| `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 3-21 summarizes the methods available through `NodeRef` interface.

**Table 3-21 Summary of NodeRef Methods; Dom Package**

| Function | Summary |
|---|---|
| NodeRef() | Constructor. |
| appendChild() | Append new child to node's list of children. |
| cloneNode() | Clone this node. |
| getAttributes() | Get attributes of this node. |
| getChildNodes() | Get children of this node. |
| getFirstChild() | Get the first child node of this node. |
| getLastChild() | Get the last child node of this node. |
| getLocalName() | Get local name of this node. |
| getNamespaceURI() | Get namespace URI of this node as a NULL-terminated string. |
| getNextSibling() | Get the next sibling node of this node. |
| getNoMod() | Tests if no modifications are allowed for this node. |
| getNodeName() | Get node's name as NULL-terminated string. |
| getNodeType() | Get DOMNodeType of the node. |
| getNodeValue() | Get node's value as NULL-terminated string. |
| getOwnerDocument() | Get the owner document of this node. |
| getParentNode() | Get parent node of this node. |
| getPrefix() | Get namespace prefix of this node. |
| getPreviousSibling() | Get the previous sibling node of this node. |
| hasAttributes() | Tests if this node has attributes. |
| hasChildNodes() | Test if this node has children. |
| insertBefore() | Insert new child into node's list of children. |
| isSupported() | Tests if specified feature is supported by the implementation. |
| markToDelete() | Sets the mark to delete the referenced node. |
| normalize() | Merge adjacent text nodes. |
| removeChild() | Remove an existing child node. |
| replaceChild() | Replace an existing child of a node. |
| resetNode() | Reset NodeRef to reference another node. |
| setNodeValue() | Set node's value as NULL-terminated string. |
| setPrefix() | Set namespace prefix of this node. |
| ~NodeRef() | 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 3-22 summarizes the methods available through `NotationRef` interface.

**Table 3-22 Summary of NotationRef Methods; Dom Package**

| Function | Summary |
|---|---|
| NotationRef() | Constructor. |
| getPublicId() | Get public ID. |
| getSystemId() | Get system ID. |
| ~NotationRef() | 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 3-23 summarizes the methods available through `ProcessingInstructionRef` interface.

**Table 3-23 Summary of ProcessingInstructionRef Methods; Dom Package**

| Function | Summary |
|---|---|
| ProcessingInstructionRef() | Constructor. |
| getData() | Get processing instruction's data. |
| getTarget() | Get processing instruction's target. |
| setData() | Set processing instruction's data. |
| ~ProcessingInstructionRef() | Public default destructor. |

## ProcessingInstructionRef()

Class constructor.

| Syntax | Description |
|---|---|
| `ProcessingInstructionRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given `ProcessingInstruction` node after a call to create `ProcessingInstruction`. |
| `ProcessingInstructionRef(`<br>`  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 3-24 summarizes the methods available through Range interface.

**Table 3-24 Summary of Range Methods; Dom Package**

| Function | Summary |
|---|---|
| CompareBoundaryPoints() | Compares boundary points. |
| cloneContent() | Makes a clone of the node. |
| cloneRange() | Clones a range of nodes. |
| deleteContents() | Deletes contents of the node. |
| detach() | Invalidate the range. |
| extractContent() | Extract the node. |
| getCollapsed() | Check if the range is collapsed. |
| getCommonAncestorContainer() | Get the deepest common ancestor node. |
| getEndContainer() | Get end container node. |
| getEndOffset() | Get offset of the end point. |
| getStartContainer() | Get start container node. |
| getStartOffset() | Get offset of the start point. |
| insertNode() | Inserts a node. |
| selectNodeContent() | Selects node content by its reference. |
| selectNode() | Selects a node. |
| setEnd() | Set end point. |
| setEndAfter() | Sets the end pointer after a specified node. |
| setEndBefore() | Set the end before a specified node. |
| setStart() | Set start point. |
| setStartAfter() | Sets start pointer after a specified node. |
| setStartBefore() | Sets start pointer before a specified node. |
| surroundContents() | Makes a node into a child of the specified node. |
| toString() | Converts an item into a string. |

## 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 3-25 summarizes the methods available through RangeException interface.

**Table 3-25 Summary of RangeException Methods; Dom Package**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | Get Oracle XML error message. |
| getRangeCode() | 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 3-26 summarizes the methods available through `TextRef` interface.

**Table 3-26 Summary of TextRef Methods; Dom Package**

| Function | Summary |
|---|---|
| TextRef() | Constructor. |
| splitText() | Split text node into two. |
| ~TextRef() | Public default destructor. |

## TextRef()

Class constructor.

| Syntax | Description |
|---|---|
| `TextRef(`<br>`  const NodeRef< Node>& node_ref,`<br>`  Node* nptr);` | Used to create references to a given text node after a call to `createtext`. |
| `TextRef(`<br>`  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 3-27 summarizes the methods available through `TreeWalker` interface.

**Table 3-27 Summary of TreeWalker Methods; Dom Package**

| Function | Summary |
|---|---|
| adjustCtx() | Attach this tree walker to another context. |
| firstChild() | Get the first child of the current node. |
| lastChild() | Get the last child of the current node. |
| nextNode() | Get the next node. |
| nextSibling() | Get the next sibling node. |
| parentNode() | Get the parent of the current node. |
| previousNode() | Get the previous node. |
| previousSibling() | 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

# 4
# Package IO APIs for C++

The IO Package for C++ APIs describes the input/output `IO` datatypes and `InputSource` methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## IO Datatypes

Table 4-1 summarizes the datatypes of the IO package.

**Table 4-1 Summary of Datatypes; IO Package**

| Datatype | Description |
|---|---|
| InputSourceType | 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 4-2 summarizes the methods available through the IO interface

**Table 4-2 Summary of IO Package Interfaces**

| Function | Summary |
|---|---|
| getBaseURI() | Get the base URI. |
| getISrcType() | Get the input source type. |
| setBaseURI() | 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; }
```

# 5
# Package Parser APIs for C++

The Parser interfaces include `Parser`datatypes, `DOMParser`methods, `GParser` methods, `ParserException` methods,`SAXHandler` methods, `SAXParser`methods, and `SchemaValidator`methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## Parser Datatypes

Table 5-1 summarizes the datatypes of the `Parser` package.

**Table 5-1 Summary of Datatypes; Parser Package**

| Datatype | Description |
|---|---|
| ParserExceptionCode | Parser implementation of exceptions. |
| DOMParserIdType | Defines parser identifiers. |
| SAXParserIdType | Defines type of node. |
| SchValidatorIdType | 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 5-2 summarizes the methods available through the `DOMParser` interface.

**Table 5-2 Summary of DOMParser Methods; Parser Package**

| Function | Summary |
|---|---|
| getContext() | Returns parser's XML context (allocation and encodings). |
| getParserId() | Get parser id. |
| parse() | Parse the document. |
| parseDTD() | Parse DTD document. |
| parseSchVal() | Parse and validate the document. |
| setValidator() | 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 5-3 summarizes the methods available through the GParser interface.

**Table 5-3 Summary of GParser Methods; Parser Package**

| Function | Summary |
|---|---|
| SetWarnDuplicateEntity() | Specifies if multiple entity declarations result in a warning. |
| getBaseURI() | Returns the base URI for the document. |
| getDiscardWhitespaces() | Checks if whitespaces between elements are discarded. |
| getExpandCharRefs() | Checks if character references are expanded. |
| getSchemaLocation() | Get schema location for this document. |
| getStopOnWarning() | Get if document processing stops on warnings. |
| getWarnDuplicateEntity() | Get if multiple entity declarations cause a warning. |
| setBaseURI() | Sets the base URI for the document. |
| setDiscardWhitespaces() | Sets if formatting whitespaces should be discarded. |
| setExpandCharRefs() | Get if character references are expanded. |
| setSchemaLocation() | Set schema location for this document. |
| setStopOnWarning() | 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 5-4 summarizes the methods available through the `ParserException` interface.

**Table 5-4 Summary of ParserException Methods; Parser Package**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | Get Oracle XML error message. |
| getParserCode() | 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 5-5 summarizes the methods available through the `SAXHandler` interface.

**Table 5-5 Summary of SAXHandler Methods; Parser Package**

| Function | Summary |
|---|---|
| CDATA() | Receive notification of CDATA. |
| XMLDecl() | Receive notification of an XML declaration. |
| attributeDecl() | Receive notification of attribute's declaration. |
| characters() | Receive notification of character data. |
| comment() | Receive notification of a comment. |
| elementDecl() | Receive notification of element's declaration. |
| endDocument() | Receive notification of the end of the document. |
| endElement() | Receive notification of element's end. |
| notationDecl() | Receive notification of a notation declaration. |
| parsedEntityDecl() | Receive notification of a parsed entity declaration. |
| processingInstruction() | Receive notification of a processing instruction. |
| startDocument() | Receive notification of the start of the document. |
| startElement() | Receive notification of element's start. |
| startElementNS() | Receive namespace aware notification of element's start. |
| unparsedEntityDecl() | Receive notification of an unparsed entity declaration. |
| whitespace() | 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 of the attribute |

| Parameter | Description |
|---|---|
| name | name of the declaration |
| 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 5-6 summarizes the methods available through the SAXParser interface.

**Table 5-6 Summary of SAXParser Methods; Parser Package**

| Function | Summary |
|---|---|
| getContext() | Returns parser's XML context (allocation and encodings). |
| getParserId() | Returns parser Id. |
| parse() | Parse the document. |
| parseDTD() | Parse the DTD. |
| setSAXHandler() | 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 5-7 summarizes the methods available through the SchemaValidator interface.

**Table 5-7 Summary of SchemaValidator Methods; Parser Package**

| Function | Summary |
|---|---|
| getSchemaList() | Return the Schema list. |
| getValidatorId() | Get validator identifier. |
| loadSchema() | Load a schema document. |
| unloadSchema() | 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 |

# 6
# Package SOAP APIs for C++

According to the W3C definition, “SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses."

The structure of a SOAP message follows:

```
[SOAP message (XML document)
   [SOAP envelope
      [SOAP header?
            element*
      ]
      [SOAP body
           (element* | Fault)?
      ]
   ]
]
```

The Oracle C++ implementation of the `SOAP` interface includes `SOAP` datatypes, `SoapException` methods, `ConnectRef` methods, and `MsgFactory` methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## SOAP Datatypes

Table 6-1 summarizes the datatypes of the `SOAP` package.

**Table 6-1 Summary of Datatypes; SOAP Package**

| Datatype | Description |
|---|---|
| SoapExceptionCode | Defines Soap-related exception codes. |
| SoapBinding | Defines binding for SOAP connections. |
| SoapRole | Defines roles for SOAP nodes. |

## SoapExceptionCode

Defines Soap-related exception codes.

```
typedef enum SoapExceptionCode {
   SOAP_UNDEFINED_ERR       = 0,
   SOAP_OTHER_ERR           = 1} SoapExceptionCode;
```

## SoapBinding

Defines binding for SOAP connections. HTTP is currently the only choice.

```
typedef enum SoapBinding {
   BIND_NONE  = 0,  /* none */
   BIND_HTTP  = 1   /* HTTP */ } SoapBinding;
```

## SoapRole

Defines roles for SOAP nodes.

```
typedef enum SoapRole {
   ROLE_UNSET = 0,  /* not specified */
   ROLE_NONE  = 1,  /* "none" */
   ROLE_NEXT  = 2,  /* "next" */
   ROLE_ULT   = 3   /* "ultimateReceiver" */ } SoapRole;
```

# SoapException Interface

Table 6-2 summarizes the methods available through the SoapException Interface.

**Table 6-2 Summary of SoapException Interfaces**

| Datatype | Description |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception |
| getMessage() | Get Oracle XML error message. |
| getMesLang() | Get current language encoding of error messages. |
| getSoapCode() | Get Soap exception code embedded in the exception. |

## getCode()

Get Oracle XML error code embedded in the exception. This is a virtual member function inherited from `XMLException`.

### Syntax

```
virtual unsigned getCode() const = 0;
```

### Returns

(unsigned) numeric error code (0 on success)

## getMessage()

Get Oracle XML error message. Virtual member function inherited from `XMLException`.

### Syntax

```
virtual oratext* getMessage() const = 0;
```

### Returns

(oratext *) Error message

## getMesLang()

Get 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

## getSoapCode()

Get SOAP exception code embedded in the exception. This is a virtual member function that defines a prototype for implementing defined member functions that return SOAP API exception codes.

### Syntax

```
virtual SoapExceptionCode getSoapCode() const = 0;
```

### Returns

(SoapExceptionCode) exception code

# ConnectRef Interface

Table 6-3 summarizes the methods available ConnectRef Interface.

**Table 6-3 Summary of ConnectRef Interfaces**

| Datatype | Description |
|---|---|
| ~ConnectRef() | Destroys a SOAP connection object |
| call() | Send a SOAP message then waits for a response. |

## ~ConnectRef()

Destructor. Destroys a SOAP connection object and frees any resources associated with it.

### Syntax

```
~ConnectRef() throw (SoapException);
```

## call()

Send a SOAP message then waits for a response.

### Syntax

```
DocumentRef< Node>* call(
   DocumentRef< Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | The SOAP message that is sent. |

### Returns

(DocumentRef) returned message

# MsgFactory Interface

Table 6-4 summarizes the methods available through the MsgFactory Interface.

**Table 6-4 Summary of MsgFactory Interfaces**

| Datatype | Description |
|---|---|
| ~MsgFactory() | Destroys the SOAP message factory instance. |
| MsgFactory() | Creates and returns a SOAP Message Factory instance. |
| addBodyElement() | Adds an element to a SOAP message body. |
| addFaultReason() | Add s an additional Reason to Fault |
| addHeaderElement() | Adds an element to SOAP header. |
| createConnection() | Creates a SOAP connection. |
| createMessage() | Creates and returns an empty SOAP message. |
| destroyMessage() | Destroyw a SOAP message. |
| getBody() | Returns a SOAP message's envelope body.. |
| getBodyElement() | Gets an element from a SOAP body. |
| getEnvelope() | Returns a SOAP part's envelope. |
| getFault() | Returns Fault code, reason, and details. |
| getHeader() | Returns a SOAP message's envelope header.. |
| getHeaderElement() | Gets an element from a SOAP header. |
| getMustUnderstand() | Gets `mustUnderstand` attribute from the SOAP header element. |
| getReasonNum() | Gets the number of reasons in `Fault` element. |
| getReasonLang() | Gets a language of a reason with a particular index. |
| getRole() | Gets role from SOAP header element. |
| hasFault() | Determines if a SOAP message contains a Fault object. |
| setFault() | Sets Fault in a SOAP message. |

**Table 6-4 (Cont.) Summary of MsgFactory Interfaces**

| Datatype | Description |
|---|---|
| setMustUnderstand() | Sets `mustUnderstand` attribute for the SOAP header element. |
| setRole() | Sets the role for SOAP header element. |

# ~MsgFactory()

Destructor. Destroys the SOAP message factory. All allocated memory is frieed and all connections are closed.

### Syntax

```
~MsgFactory() throw (SoapException);
```

# MsgFactory()

Creates and returns a SOAP Message Factory instance.

### Syntax

```
MsgFactory(
   Context* ctx_ptr) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `ctx_ptr` | `TContext` object |

### Returns

(MsgFactory) object

# addBodyElement()

Adds an element to a SOAP message body.

### Syntax

```
Node* addBodyElement( DocumentRef< Node>& msg,
   oratext *qname,
   oratext *uri) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | SOAP message |
| `qname` | QName of element to add |
| `uri` | Namespace URI of element to add |

### Returns

(Node*) pointer to the created element

## addFaultReason()

Adds an additional Reason to Fault. The same re son text may be provided in different languages. When the fault is created, the primary language and reasonare added at that time; use this function to add additional translations of the reason.

### Syntax

```
void addFaultReason(
   DocumentRef< Node>& msg,
   oratext *reason,
   oratext *lang) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | SOAP message |
| `reason` | Human-readable fault reason |
| `lang` | Language of reason |

## addHeaderElement()

Adds an element to SOAP header.

### Syntax

```
Node* addHeaderElement(
   DocumentRef< Node>& msg,
   oratext *qname, oratext *uri) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | SOAP message |
| `qname` | QName of element to add |
| `uri` | Namespace URI of element to add |

### Returns

(Node*) pointer to the created element

## createConnection()

Creates a SOAP connection object. The connection reference object should explicitly deleted by the user.

### Syntax

```
ConnectRef< Node>* createConnection(
   SoapBinding bind,
   void *endp,
   oratext *buf,
   ubig_ora bufsiz,
   oratext *msgbuf,
   ubig_ora msgbufsiz) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `bind` | connection binding |
| `endp` | connection endpoint |
| `buf` | data buffer (or `NULL` to have one allocated) |
| `bufsiz` | size of data buffer (or 0 for default size) |
| `msgbuf` | message buffer (or `NULL` to have one allocated) |
| `msgfubsiz` | size of message buffer (or 0 for default size) |

### Returns

(ConnectRef) connect object

## createMessage()

Creates and returns an empty SOAP message. The reference object should be explicitly deleted by the user when no longer needed.

### Syntax

```
DocumentRef< Node>* CreateMessage() throw (SoapException);
```

### Returns

(DocumentRef*) SOAP message, or an exception

## destroyMessage()

Destroys a SOAP message.

### Syntax

```
void destroyMessage(
   DocumentRef< Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | The SOAP message. |

## getBody()

Returns a SOAP message's envelope body as a pointer to the body's element node.

### Syntax

```
Node* getBody(
   DocumentRef<Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |

### Returns

(Node*) pointer to the SOAP message's envelope body

## getBodyElement()

Gets an element from a SOAP body as a pointer to its element node.

### Syntax

```
Node* getBodyElement(
   DocumentRef< Node>& msg,
   oratext *uri,
   oratext *local) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |
| uri | Namespace URI of element to add |
| local | Local name of element to get |

### Returns

(Node*) pointer to the named element

## getEnvelope()

Returns a SOAP part's envelope as a pointer to envelope element node.

### Syntax

```
Node* getEnvelope(
   DocumentRef<Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |

### Returns

(Node*) pointer to the SOAP message's envelope

## getFault()

Returns Fault code, reason, and details through user variables. NULL may be supplied for any part that is not needed. For lang, if the pointed-to variable is NULL, it will be set to the default language, that of the first reason.

### Syntax

```
Node* getFault(
   DocumentRef< Node>& msg,
   oratext **code,
   oratext **reason,
   oratext **lang,
   oratext **node,
   oratext **role) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |
| code | Fault code |
| reason | Human-readable fault reason |
| lang | Desired reason language or NULL (default language, same as for the first reason) |
| node | Fault node |
| role | Role: next, none, or ulitmate receiver (not used in 1.1) |

### Returns

(Node) pointer to the detail

## getHeader()

Returns a SOAP message's envelope header as a pointer to the header element node.

### Syntax

```
Node* getHeader(
   DocumentRef< Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |

### Returns

(Node*) pointer to the SOAP message's envelope header

## getHeaderElement()

Gets an element from a SOAP header as a pointer to its element node.

### Syntax

```
Node* getHeaderElement(
   DocumentRef< Node>& msg,
   oratext *uri,
   oratext *local) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |
| uri | Namespace URI of element to get |
| local | Local name of element to get |

### Returns

(Node*) pointer to the named element

## getMustUnderstand()

Gets `mustUnderstand` attribute from SOAP header element.

### Syntax

```
boolean getMustUnderstand(
   ElementRef< Node>& elem) throw (SoapException);
```

| Parameter | Description |
|---|---|
| elem | SOAP header element |

### Returns

(boolean) value of the `mustUnderstand` attribute

## getReasonNum()

Determines the number of reasons in the `Fault` element. Returns `0` if no `Fault`.

### Syntax

```
ub4 getReasonNum(
   DocumentRef< Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | SOAP message |

### Returns

(up4) number of reasons in Fault element

## getReasonLang()

Returns the language of a reason with a particular index.

### Syntax

```
oratext* getReasonLang(
   DocumentRef< Node>& msg,
   ub4 idx) throw (SoapException);
```

| Parameter | Description |
|---|---|
| `msg` | SOAP message |
| `idx` | index of a fault reason |

### Returns

(oratext *) value of property or `NULL`.

## getRole()

Gets role from SOAP header element.

### Syntax

```
SoapRole getRole(
   ElementRef< Node>& elem) throw (SoapException);
```

| Parameter | Description |
|---|---|
| elem | Reference to the header element |

**Returns**

(SoapRole) header element's role

## hasFault()

Determines if a SOAP message contains a Fault object.

**Syntax**

```
boolean hasFault(
   DocumentRef< Node>& msg) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |

**Returns**

(boolean) TRUE if there's a Fault, FALSE if not

## setFault()

Sets Fault in a SOAP message.

**Syntax**

```
void setFault(
   DocumentRef< Node>& msg,
   oratext *node,
   oratext *code,
   oratext *reason,
   oratext *lang,
   oratext *role,
  ElementRef< Node>& detail) throw (SoapException);
```

| Parameter | Description |
|---|---|
| msg | SOAP message |
| node | URI of SOAP node which faulted |
| code | Fault code |
| reason | Human-readable fault reason |

| Parameter | Description |
|---|---|
| lang | Language |
| role | URI representing role, Role (1.2), unused (1.1) |
| detail | User-defined elements |

## setMustUnderstand()

Sets mustUnderstand attribute for the SOAP header element.

### Syntax

```
void setMustUnderstand(
   ElementRef< Node>& elem,
   boolean mustUnderstand) throw (SoapException);
```

| Parameter | Description |
|---|---|
| elem | SOAP header element |
| mustUnderstand | mustUnderstand value (TRUE or FALSE) |

## setRole()

Sets the role for SOAP header element.

### Syntax

```
void setRole(
   ElementRef< Node>& elem,
   SoapRole role) throw (SoapException);
```

| Parameter | Description |
|---|---|
| elem | reference to the header element |
| role | role value |

7

# Package Tools APIs for C++

The `Tools` package contains types and methods for creating and instantiating Oracle XML tools. It describes `Tools` datatypes, `Factory` methods, and `FactoryException` methods.

## Tools Datatypes

Table 7-1 summarizes the datatypes of the `Tools` package.

**Table 7-1 Summary of Datatypes; Tools Package**

| Datatype | Description |
|---|---|
| FactoryExceptionCode | Tool Factory exceptions. |

### FactoryExceptionCode

Tool Factory exceptions.

**Definition**

```
typedef enum FactoryExceptionCode {
   FACTORY_UNDEFINED_ERR = 0,
   FACTORY_OTHER_ERR = 1
} FactoryExceptionCode;
```

## Factory Interface

Table 7-2 summarizes the methods available through the `Factory` interface.

**Table 7-2 Summary of Factory Methods; Tools Package**

| Function | Summary |
|---|---|
| Factory() | Constructor. |
| createDOMParser() | Create DOM Parser. |
| createSAXParser() | Create SAX Parser. |
| createSchemaValidator() | Create schema validator. |
| createXPathCompProcessor() | Create extended XPath processor. |
| createXPathCompiler() | Create XPath compiler. |
| createXPathProcessor() | Create XPath processor. |
| createXPointerProcessor() | Create XPointer processor. |
| createXslCompiler() | Create Xsl compiler. |
| createXslExtendedTransformer() | Create XSL extended transformer. |

**Table 7-2 (Cont.) Summary of Factory Methods; Tools Package**

| Function | Summary |
|---|---|
| createXslTransformer() | Create XSL transformer. |
| getContext() | Get factory's context. |
| ~Factory() | Default destructor. |

# Factory()

Class constructor.

| Syntax | Description |
|---|---|
| `Factory()`<br>`throw (FactoryException);` | Default constructor |
| `Factory(`<br>`  Context* ctx_ptr)`<br>`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 7-3 summarizes the methods available through the `FactoryException` interface.

**Table 7-3 Summary of FactoryException Methods; Tools Package**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getFactoryCode() | Get FactoryException code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | 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

# 8
# Package XPath APIs for C++

The `XPath` package for C++ contains types and interfaces related to `XPath` processing, such as `XPath` datatypes, `CompProcessor` methods,`Compiler` methods, `NodeSet` methods, `Processor` methods, `XPathException` methods, and `XPathObject` methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## XPath Datatypes

Table 8-1 summarizes the datatypes of the `XPath` package.

**Table 8-1 Summary of Datatypes; XPath Package**

| Datatype | Description |
|---|---|
| XPathCompIdType | Defines XPath compiler identifiers. |
| XPathObjType | Defines object types for XPath 1.0 based implementations. |
| XPathExceptionCode | XPath related exception codes. |
| XPathPrIdType | 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 8-2 summarizes the methods available through the `CompProcessor` interface.

**Table 8-2 Summary of CompProcessor Methods; XPath Package**

| Function | Summary |
|---|---|
| getProcessorId() | Get processor's Id. |
| process() | Evaluate XPath expression against given document. |
| processWithBinXPath() | 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 8-3 summarizes the methods available through the `Compiler` interface.

**Table 8-3 Summary of Compiler Methods; XPath Package**

| Function | Summary |
|---|---|
| compile() | Compile XPath and return its compiled binary representation. |
| getCompilerId() | 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 8-4 summarizes the methods available through the NodeSet interface.

**Table 8-4 Summary of NodeSet Methods; XPath Package**

| Function | Summary |
|---|---|
| getNode() | Get node given its index. |
| getSize() | 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 8-5 summarizes the methods available through the Processor interface.

**Table 8-5 Summary of Processor Methods; XPath Package**

| Function | Summary |
|---|---|
| getProcessorId() | Get processor's Id. |
| process() | 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 |
| `xpath_exp` | XPath expression |

**Returns**

`(XPathGenObject*)` XPath object

# XPathException Interface

Table 8-6 summarizes the methods available through the `XPathException` interface.

**Table 8-6 Summary of XPathException Methods; XPath Package**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | Get Oracle XML error message. |
| getXPathCode() | 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 8-7 summarizes the methods available through the `XPathObject` interface.

**Table 8-7 Summary of XPathObject Methods; XPath Package**

| Function | Summary |
|---|---|
| XPathObject() | Copy constructor. |
| getNodeSet() | Get the node set. |
| getObjBoolean() | Get boolean from object. |
| getObjNumber() | Get number from object. |
| getObjString() | Get string from object. |
| getObjType() | 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;
```

# 9
# Package XPointer APIs for C++

The `XPointer` package for C++ contains types and methods related to `XPointer` processing, such as `XPointer` datatypes, `Processor` methods, `XppException` methods, `XPPLocation` methods, and `XppLocSet` methods.

## XPointer Datatypes

Table 9-1 summarizes the datatypes of the `XPointer` package.

**Table 9-1 Summary of Datatypes; XPointer Package**

| Datatype | Description |
|---|---|
| XppExceptionCode | Defines XPath compiler identifiers. |
| XppPrIdType | Defines XPointer processor identifiers. |
| XppLocType | 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 9-2 summarizes the methods available through the `Processor` interface.

**Table 9-2 Summary of Processor Methods; XPointer Package**

| Function | Summary |
|---|---|
| getProcessorId() | Get processor's Id. |
| process() | 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;
```

**Table 9-3 Description of Process Method Parameters**

| Parameter | Description |
|---|---|
| `isrc_ptr` | instance document to process |
| `xpp_exp` | XPointer expression |

### Returns

`(XppLocSet*)` XPath object

# XppException Interface

Table 9-4 summarizes the methods available through the `XPPException` interface.

**Table 9-4 Summary of XppException Methods; Package XPointer**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMesLang() | Get Oracle XML error message. |
| getXppCode() | 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 9-5 summarizes the methods available through the `XppLocation` interface.

**Table 9-5 Summary of XppLocation Methods; XPointer Package**

| Function | Summary |
|---|---|
| getLocType() | Get the location type. |
| getNode() | Get the node. |
| getRange() | 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 9-6 summarizes the methods available through the `XppLocSet` interface.

**Table 9-6 Summary of XppLocSet Methods; XPointer Package**

| Function | Summary |
|---|---|
| getItem() | Get item given its index. |
| getSize() | Get location set size. |

## getItem()

Returns a reference to the item.

### Syntax

```
XppLocation< Node>* getItem(
   ub4 index) const;
```

**Table 9-7 Summary of getItem Method; XPointer Package**

| 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

# 10
# Package Xsl APIs for C++

The `Xsl` package for C++ contains types and methods related to XSL transformation. They include `Xsl` datatypes, `Compiler` methods, `CompTransformer` methods, `Transformer` methods, and `XSLException` methods.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide*

## Xsl Datatypes

Table 10-1 summarizes the datatypes of the `Xsl` package.

**Table 10-1 Summary of Datatypes; Xsl Package**

| Datatype | Description |
|---|---|
| XslCompIdType | Defines XSL compiler identifiers. |
| XslExceptionCode | Defines XSLT related exceptions. |
| XslTrIdType | 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 10-2 summarizes the methods available through the `Compiler` interface.

**Table 10-2 Summary of Compiler Methods; Xsl Package**

| Function | Summary |
|---|---|
| compile() | Compile Xsl and return its compiled binary representation. |
| getCompilerId() | Get compiler's Id. |
| getLength() | 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 10-3 summarizes the methods available through the `CompTransformer` interface.

**Table 10-3 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(<br>nputSource* isrc_ptr)<br>throw (XslException) = 0; | Transform the document and return DOM. |

| Syntax | Description |
|---|---|
| `virtual void transform(`<br>`  InputSource* isrc_ptr,`<br>`  SAXHandlerRoot* hdlr_ptr)`<br>`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 10-4 summarizes the methods available through the `Transformer` interface.

**Table 10-4 Summary of Transformer Methods; Xsl Package**

| Function | Summary |
|---|---|
| getTransformerId() | Get transformer's Id. |
| setSAXHandler() | Set SAX handler. |
| setXSL() | Set XSLT document for this transformer. |
| transform() | 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(`<br>`  nputSource* isrc_ptr)`<br>`throw (XslException) = 0;` | Transform the document and return DOM. |
| `virtual void transform(`<br>`  InputSource* isrc_ptr,`<br>`  SAXHandlerRoot* hdlr_ptr)`<br>`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 10-5 summarizes the methods available through the XSLException interface.

**Table 10-5 Summary of XSLException Methods; Xsl Package**

| Function | Summary |
|---|---|
| getCode() | Get Oracle XML error code embedded in the exception. |
| getMesLang() | Get current language (encoding) of error messages. |
| getMessage() | Get Oracle XML error message. |
| getXslCode() | 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

# Index

## Symbols



## A



## C

## D

## E



## F

## G

## H



## I



## L



## M



## N

## O



## P

## R



## S

## T



## U



## W



## X