

D

The XML Document Object Model

This appendix lists the interfaces in the Document Object Model (DOM) Level 3. Examples showing how to use some of these interfaces appear in Chapter 11.

Unfortunately, the DOM Working Group defines too many “modules” for DOM functionality to be covered in this one appendix. In fact, at the time of writing, the W3C’s site listed the following seven Technical Reports for different types of DOM activities:

- ❑ The Core interfaces, which are the base set of interfaces used for working with HTML and XML documents
- ❑ The Load and Save interfaces, which are used to load XML documents into a DOM (from a file, URI, stream, etc.) or save an XML document from a DOM (to a file, URI, stream, etc.)
- ❑ The Validation interfaces, which are used to ensure that an XML document is valid, per its schema document(s)
- ❑ The XPath interfaces, for accessing a DOM tree using XPath syntax
- ❑ The Views and Formatting interfaces, which can be used to dynamically access and modify a document’s structure, style, and contents
- ❑ The Events interfaces, which allow for event handlers
- ❑ The Abstract Schemas interfaces, which allow an interface to schema documents (DTD and XML Schema)

In addition, there is another Technical Report on “DOM Requirements,” which doesn’t specifically list interfaces.

At the time of writing, only the Core, Load and Save, and Validation modules were full W3C Recommendations, so these are the modules covered in this appendix. Luckily, these are the ones that you are most likely to access in day-to-day work with the DOM.

Appendix D: The XML Document Object Model

This appendix provides a handy guide to the DOM interfaces, but if you'd like further information, you can always go to the W3C's website to view the actual recommendations:

- ❑ Core: <http://www.w3.org/TR/DOM-Level-3-Core>
- ❑ Load and Save: <http://www.w3.org/TR/DOM-Level-3-LS>
- ❑ Validation: <http://www.w3.org/TR/DOM-Level-3-Val>

The interfaces are illustrated in the figures in the following section.

Notation

The notation used for the DOM interfaces is Interface Definition Language (IDL). For example, a property named `length` that returns an integer value might be defined in the DOM Recommendation as follows:

```
readonly attribute unsigned long length;
```

This appendix uses the following, more friendly approach:

Property	Type	Description
<code>length</code>	unsigned long (read-only)	A description of the property would go here

If you're not familiar with terms such as *unsigned long* or *unsigned short*, don't worry; just think of either of these as an integer — that is, a number that doesn't have a decimal. Unless you're writing a DOM implementation yourself, it doesn't matter too much how big the numbers are for the purpose of this appendix.

Figure D-1 shows the DOM Core interfaces.

Appendix D: The XML Document Object Model

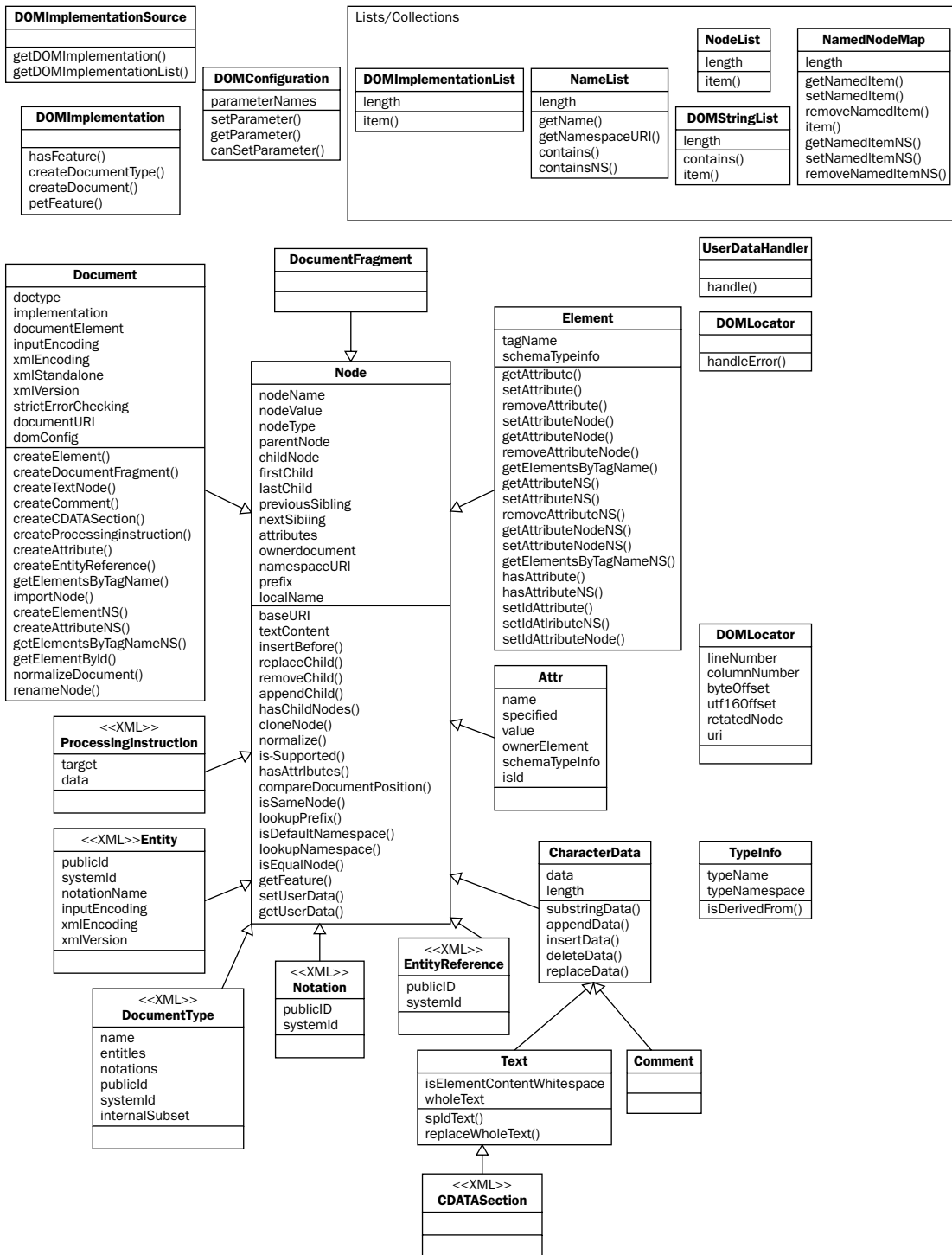


Figure D-1

Appendix D: The XML Document Object Model

Figure D-2 shows the DOM Load and Save interfaces.

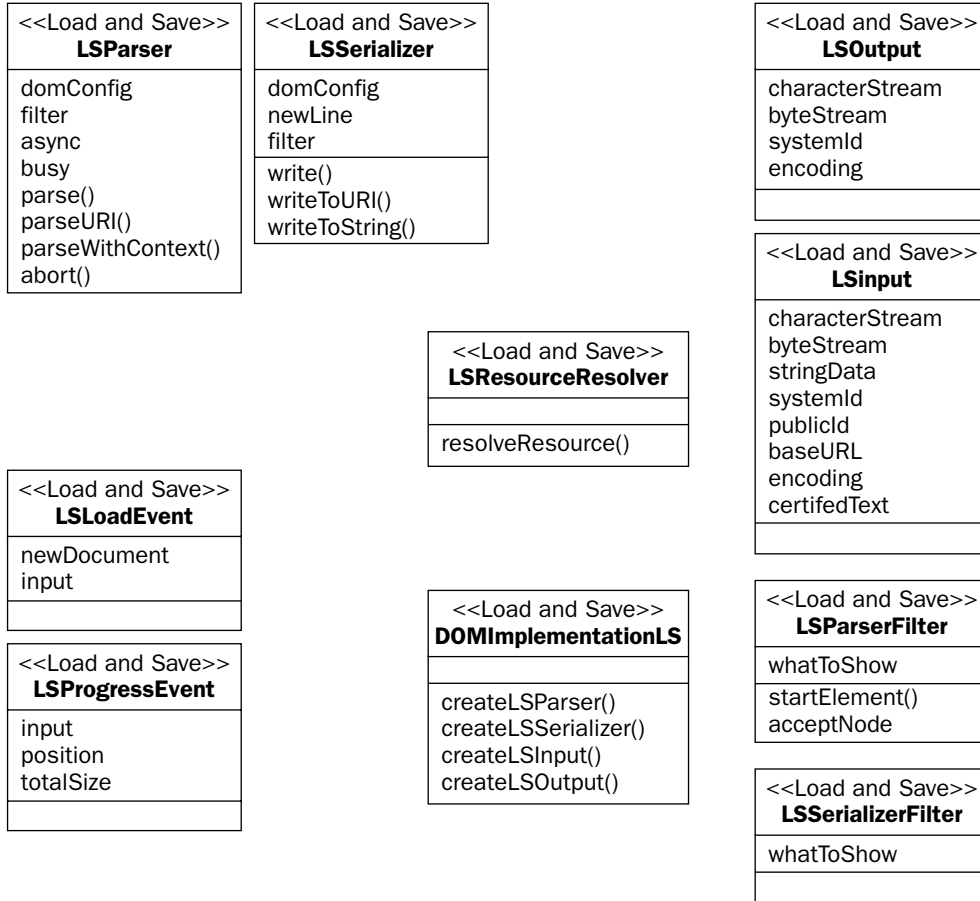


Figure D-2

Figure D-3 shows the DOM Validation interfaces.

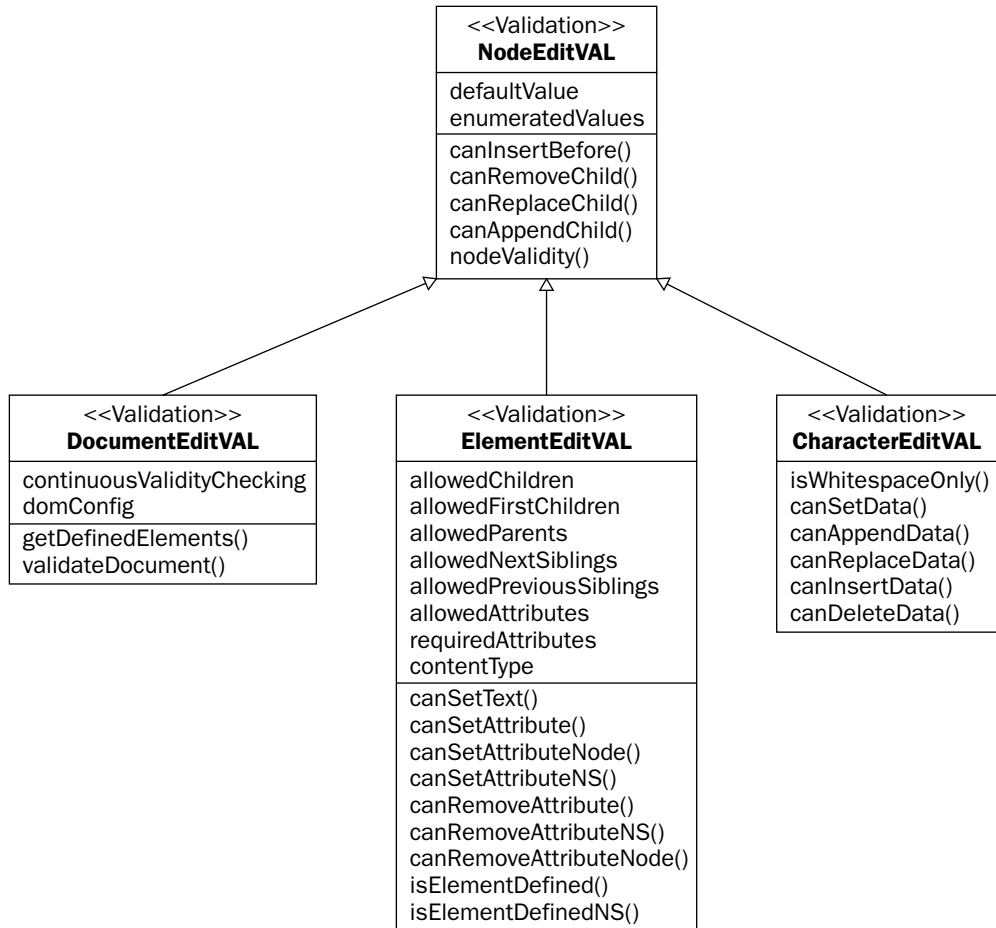


Figure D-3

Basic Datatypes

Because the DOM is language- and platform-independent, there are some inherent difficulties in making it generic. For example, you can't just specify that a particular property or method will return a "string," because the different programming languages used to implement the DOM have a different idea of what a "string" is.

To get around this, the DOM Recommendation(s) specify some basic datatypes that are used throughout the interface descriptions. DOM implementers can create their own objects to implement these datatypes, or simply use built-in ones. (For example, Java programmers could use `String` instead of `DOMString`, as a Java `String` is the same as a `DOMString`, as specified by the DOM Recommendation.)

Strings

To ensure interoperability, the DOM specifies a `DOMString` datatype, which is a sequence of 16-bit characters. These characters must be in the UTF-16 encoding.

However, even though the DOM Recommendation always uses this `DOMString` datatype, the actual datatype used by a programming language may be an inherent string datatype. For example, in Java, a DOM implementation would use a normal `String`, as would a Visual Basic DOM implementation, because both Java and VB strings are UTF-16.

This appendix uses the `DOMString` type for consistency.

User Data

The DOM Level 3 introduces the concept of *user data*, which can be assigned to a node. This information is not part of the XML document, but it is considered useful to the programmer.

The `DOMUserData` datatype is used to store this user-defined data. It is defined as using the `any` datatype, meaning that it is very generic. In Java, it would probably be represented by the `Object` datatype; in Visual Basic, probably by the `Variant` datatype.

Objects

There are various places in the DOM Recommendations where a method or property simply returns some type of object, which doesn't necessarily implement any of the interfaces defined. In this case, the `DOMObject` type is used.

Again, this would be very similar to the `Object` datatype in Java.

Fundamental Interfaces: Core Module

The DOM Fundamental Interfaces are interfaces that *all* DOM implementations must provide, even if they aren't designed to work with XML documents.

DOMException

An object implementing the `DOMException` interface is thrown whenever an error occurs in the DOM.

Property	Type	Description
<code>code</code>	unsigned short	Represents the <i>exception code</i> this <code>DOMException</code> is reporting

The `code` property can take the following values:

Appendix D: The XML Document Object Model

Exception Code	Value	Description
INDEX_SIZE_ERR	1	The index or size is negative or greater than the allowed value
DOMSTRING_SIZE_ERR	2	The specified range of text does not fit into a DOMString
HIERARCHY_REQUEST_ERR	3	The node is inserted somewhere it doesn't belong
WRONG_DOCUMENT_ERR	4	The node is used in a document other than the one that created it, and that document doesn't support it
INVALID_CHARACTER_ERR	5	A character was passed that is not valid in XML
NO_DATA_ALLOWED_ERR	6	Data was specified for a node that does not support data
NO_MODIFICATION_ALLOWED_ERR	7	An attempt was made to modify an object that doesn't allow modifications
NOT_FOUND_ERR	8	An attempt was made to reference a node that does not exist
NOT_SUPPORTED_ERR	9	The implementation does not support the type of object requested
INUSE_ATTRIBUTE_ERR	10	An attempt was made to add a duplicate attribute
INVALID_STATE_ERR	11	An attempt was made to use an object that is not, or is no longer, usable
SYNTAX_ERR	12	An invalid or illegal string was passed
INVALID_MODIFICATION_ERR	13	An attempt was made to modify the type of the underlying object
NAMESPACE_ERR	14	An attempt was made to create or change an object in a way that is incompatible with namespaces
INVALID_ACCESS_ERR	15	A parameter was passed or an operation attempted that the underlying object does not support
VALIDATION_ERR	16	The XML document was modified in such a way that it would become invalid
TYPE_MISMATCH_ERR	17	A parameter was passed to a DOM method that wasn't the correct type; for example, an Element was passed when an Attr was expected.

DOMError

This interface describes an error. There are no methods, just attributes.

Property	Type	Description
severity	unsigned short (read-only)	The severity of the error, as defined in the following table
message	DOMString (read-only)	A string describing the type of error that occurred
type	DOMString (read-only)	A string indicating which related data is expected in the <code>relatedData</code>
relatedException	DOMObject (read-only)	The related platform-dependent exception, if any
relatedData	DOMObject (read-only)	Related dependent data, if any
location	DOMLocator (read-only)	The location of the error

ErrorSeverity Constant	Description
SEVERITY_WARNING	A “warning,” meaning that the parser found a problem with the XML document, but it is not severe enough to be an error or fatal error
SEVERITY_ERROR	An “error” whereby the XML document violates the rules in the XML recommendation; results are undefined
SEVERITY_FATAL_ERROR	A “fatal error” whereby the XML parser must stop processing the XML document (except to find additional errors)

DOMErrorHandler

This is a callback interface that the DOM implementation can call when it comes across an error while processing a document. In other words, you can implement this interface yourself, in your own code; whenever your DOM implementation comes across an error, it will call your object, which implements this interface so that you can handle it.

Method	Description
boolean <code>handleError(DOMError error)</code>	This method is called by the DOM implementation whenever an error occurs. If your code returns <code>true</code> and the error is not a “fatal error,” it means that you’re indicating to the DOM implementation that it should continue to try parsing the XML document.

Node

The `Node` interface is the base interface upon which most of the DOM objects are built, and contains methods and attributes that can be used for all types of nodes. The interface also includes some helper methods and attributes that only apply to particular types of nodes.

Remember that any part of an XML document — an element, an attribute, a piece of text, a processing instruction, and so on — is considered a “node,” so this interface is very generic.

Property	Type	Description
<code>nodeName</code>	<code>DOMString</code> (read-only)	The name of the node. Will return different values, depending on the <code>nodeType</code> , as listed in the next table.
<code>nodeValue</code>	<code>DOMString</code>	The value of the node. Will return different values, depending on the <code>nodeType</code> , as listed in the next table.
<code>nodeType</code>	unsigned short (read-only)	The type of node. Will be one of the values from the next table.
<code>parentNode</code>	<code>Node</code> (read-only)	The node that is this node’s parent
<code>childNodes</code>	<code>NodeList</code> (read-only)	A <code>NodeList</code> containing all of this node’s children. If there are no children, an empty <code>NodeList</code> is returned, not <code>NULL</code> .
<code>firstChild</code>	<code>Node</code> (read-only)	The first child of this node. If there are no children, this returns <code>NULL</code> .
<code>lastChild</code>	<code>Node</code> (read-only)	The last child of this node. If there are no children, this returns <code>NULL</code> .
<code>previousSibling</code>	<code>Node</code> (read-only)	The node immediately preceding this node. If there is no preceding node, this returns <code>NULL</code> .
<code>nextSibling</code>	<code>Node</code> (read-only)	The node immediately following this node. If there is no following node, this returns <code>NULL</code> .
<code>attributes</code>	<code>NamedNodeMap</code> (read-only)	A <code>NamedNodeMap</code> containing the attributes of this node. If the node is not an element, this returns <code>NULL</code> .
<code>ownerDocument</code>	<code>Document</code> (read-only)	The document to which this node belongs
<code>namespaceURI</code>	<code>DOMString</code> (read-only)	The namespace URI of this node. Returns <code>NULL</code> if a namespace is not specified.
<code>prefix</code>	<code>DOMString</code>	The namespace prefix of this node. Returns <code>NULL</code> if a namespace is not specified.
<code>localName</code>	<code>DOMString</code> (read-only)	Returns the local part of this node’s <code>QName</code>

Table continued on following page

Appendix D: The XML Document Object Model

Property	Type	Description
baseURI	DOMString (read-only)	This property returns the node's "base URI," as defined in the XML Information Set Recommendation. This recommendation is beyond the scope of this book, but suffice it to say that a base URI allows namespace names to use <i>relative paths</i> ; for example, if I specify a base URI of <code>http://www.wiley.com</code> , I might declare my namespace to be <code>/pers</code> and the XML parser would figure out that the full namespace name is actually <code>http://www.wiley.com/pers</code> . You can find more information at http://www.w3.org/TR/xmlbase/ and http://www.w3.org/TR/xml-infoset/ .
textContent	DOMString	Returns the text content of this node, <i>and any descendent nodes</i> . When this property is set, be aware that any existing children of the node will be removed and replaced with a <code>Text</code> node containing the value you set it to.

The values of the `nodeName` and `nodeValue` properties depend on the value of the `nodeType` property, which can return one of the following constants:

nodeType Property Constant	nodeName	NodeValue
ELEMENT_NODE	Tag name	NULL
ATTRIBUTE_NODE	Name of attribute	Value of attribute
TEXT_NODE	#text	Content of the text node
CDATA_SECTION_NODE	#cdata-section	Content of the CDATA section
ENTITY_REFERENCE_NODE	Name of entity referenced	NULL
ENTITY_NODE	Entity name	NULL
PROCESSING_INSTRUCTION_NODE	Target	Entire content excluding the target
COMMENT_NODE	#comment	Content of the comment
DOCUMENT_NODE	#document	NULL
DOCUMENT_TYPE_NODE	Document type name	NULL
DOCUMENT_FRAGMENT_NODE	#document-fragment	NULL
NOTATION_NODE	Notation name	NULL

Appendix D: The XML Document Object Model

The position of a node is specified using a `DocumentPosition`, which can be one of the following:

DocumentPosition Constant	Description
<code>DOCUMENT_POSITION_CONTAINED_BY</code>	The node is contained by the reference node
<code>DOCUMENT_POSITION_CONTAINS</code>	The node contains the reference node
<code>DOCUMENT_POSITION_DISCONNECTED</code>	The nodes are not connected
<code>DOCUMENT_POSITION_FOLLOWING</code>	The node follows the reference node
<code>DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC</code>	Whether this node precedes or follows the reference node is implementation-specific
<code>DOCUMENT_POSITION_PRECEDING</code>	The reference node follows this node

Method	Description
<code>Node insertBefore(Node newChild, Node refChild)</code>	Inserts the <code>newChild</code> node before the existing <code>refChild</code> . If <code>refChild</code> is <code>NULL</code> , it inserts the node at the end of the list. Returns the inserted node.
<code>Node replaceChild(Node newChild, Node oldChild)</code>	Replaces <code>oldChild</code> with <code>newChild</code> . Returns <code>oldChild</code> .
<code>Node removeChild(Node oldChild)</code>	Removes <code>oldChild</code> from the list and returns it
<code>Node appendChild(Node newChild)</code>	Adds <code>newChild</code> to the end of the list and returns it
<code>boolean hasChildNodes()</code>	Returns a Boolean: <code>true</code> if the node has any children, <code>false</code> otherwise
<code>Node cloneNode(boolean deep)</code>	Returns a duplicate of this node. If the boolean <code>deep</code> parameter is <code>true</code> , this will recursively clone the subtree under the node; otherwise, it will only clone the node itself.
<code>void normalize()</code>	If there are multiple adjacent <code>Text</code> child nodes (from a previous call to <code>Text.splitText()</code>), this method will combine them again. It doesn't return a value.
<code>boolean isSupported(DOMString feature, DOMString version)</code>	Indicates whether this implementation of the DOM supports the <code>feature</code> passed. Returns a Boolean: <code>true</code> if it supports the feature, <code>false</code> otherwise.
<code>boolean hasAttributes()</code>	Returns <code>true</code> if this node has attributes, or <code>false</code> otherwise

Table continued on following page

Appendix D: The XML Document Object Model

Method	Description
unsigned short compareDocumentPosition(Node other)	Compares this node with the one passed in the <code>other</code> parameter, to determine the relative position of the two nodes. The return value is one of the values of the <code>DocumentPosition</code> constants.
boolean isSameNode(Node other)	Returns <code>true</code> if the <code>Node</code> object passed in <code>other</code> is the same as this one — in other words, if there are two references to the same object This is different from the case where there are two distinct nodes that happen to have the same attributes, text children, etc; that condition would be tested using the <code>isEqualNode()</code> method.
DOMString lookupPrefix (DOMString namespaceURI)	Looks up the namespace prefix associated with the passed namespace URI. Default namespace declarations are ignored by this method.
boolean isDefaultNamespace (DOMString namespaceURI)	Returns <code>true</code> if the specified namespace URI is the default namespace for the document, <code>false</code> otherwise
DOMString lookupNamespaceURI (DOMString prefix)	Returns the namespace URI associated with the prefix passed.
boolean isEqualNode(Node arg)	Returns <code>true</code> if the node passed in the <code>arg</code> parameter is equal to this node. In other words, the other node has the same namespace URI and local name, same <code>nodeValue</code> , same attributes and values (if applicable), etc. If any of this differs between the two nodes, <code>false</code> is returned. If the two <code>Node</code> objects point to the same in-memory object, <code>true</code> would be returned, just as it would from the <code>isSameNode()</code> method.
DOMObject getFeature (DOMString feature, DOMString version)	Returns a specialized object that implements the API(s) specified by the <code>feature</code> and <code>version</code> parameters
DOMUserData setUserData (DOMString key, DOMUserData data)	Associates a <code>DOMUserData</code> object with the specified key for this node. The value can later be retrieved using the <code>getUserData()</code> method.
DOMUserData getUserData (DOMString key)	Returns the <code>DOMUserData</code> object specified by the <code>key</code> attribute. If there is no object for the specified key, <code>NULL</code> is returned.

Document

An object implementing the `Document` interface represents the entire XML document. This object is also used to create other nodes at runtime. The `Document` interface extends the `Node` interface.

Property	Type	Description
<code>doctype</code>	<code>DocumentType</code> (read-only)	Returns a <code>DocumentType</code> object indicating the document type associated with this document. If the document has no document type specified, it returns <code>NULL</code> .
<code>implementation</code>	<code>DOMImplementation</code> (read-only)	The <code>DOMImplementation</code> object used for this document
<code>documentElement</code>	<code>Element</code> (read-only)	The root element for this document
<code>inputEncoding</code>	<code>DOMString</code> (read-only)	The encoding that was used for this document at the time of parsing. Returns <code>NULL</code> if the input encoding is not known.
<code>xmlEncoding</code>	<code>DOMString</code> (read-only)	The encoding of this document, as specified in the XML declaration. Returns <code>NULL</code> when not known.
<code>xmlStandalone</code>	<code>boolean</code>	Returns whether this document is “standalone,” as specified in the XML declaration. Returns <code>false</code> if this value was not set on the XML declaration. It should also be noted, per the DOM Recommendation, that this attribute is not validated; the DOM will simply return whatever was specified in the XML declaration.
<code>xmlVersion</code>	<code>DOMString</code>	The XML version number specified in the XML declaration. If there is no XML declaration but the document supports the XML feature — in other words, if it is XML — then <code>1.0</code> is returned. For non-XML documents (such as HTML), this attribute returns <code>NULL</code> .
<code>strictErrorChecking</code>	<code>boolean</code>	Indicates whether the DOM implementation should enforce error checking. (Set to <code>true</code> by default.)
<code>documentURI</code>	<code>DOMString</code>	The location of the document. If the document’s location is not known — or if the document was created programmatically, rather than retrieved from a URI — then <code>NULL</code> is returned.
<code>domConfig</code>	<code>DOMConfiguration</code> (read-only)	Returns the configuration used when the <code>normalizeDocument()</code> method is called

Appendix D: The XML Document Object Model

Method	Description
Element createElement (DOMString tagName)	Creates an element with the name specified
DocumentFragment createDocumentFragment()	Creates an empty DocumentFragment object
Text createTextNode (DOMString data)	Creates a Text node, containing the text in data
Comment createComment (DOMString data)	Creates a Comment node, containing the text in data
CDATASection createCDATASection (DOMString data)	Creates a CDATASection node, containing the text in data
ProcessingInstruction createProcessingInstruction (DOMString target, DOMString data)	Creates a ProcessingInstruction node, with the specified target and data
Attr createAttribute(DOMString name)	Creates an attribute, with the specified name
EntityReference createEntityReference (DOMString name)	Creates an entity reference, with the specified name
NodeList getElementsByTagName (DOMString tagName)	Returns a NodeList of all elements in the document with this tagName. The elements are returned in document order.
Node importNode(Node importedNode, boolean deep)	Imports importedNode from another document into this one. The original node is not removed from the old document; it is just cloned. (The boolean deep parameter specifies whether it is a <i>deep</i> or <i>shallow</i> clone: deep means the subtree under the node is also cloned; shallow means only the node itself is cloned.) Returns the new node.
Element createElementNS (DOMString namespaceURI, DOMString qualifiedName)	Creates an element, with the specified namespace and QName
Attr createAttributeNS(DOMString namespaceURI, DOMString qualifiedName)	Creates an attribute, with the specified namespace and QName
NodeList getElementsByTagNameNS (DOMString namespaceURI, DOMString localName)	Returns a NodeList of all the elements in the document that have the specified local name and are in the namespace specified by namespaceURI
Element getElementById (DOMString elementID)	Returns the element with the ID specified in elementID. If there is no such element, it returns NULL.

Method	Description
<code>normalizeDocument()</code>	This method causes the DOM implementation to “normalize” itself. For example, any contiguous <code>Text</code> nodes would be grouped together into one <code>Text</code> node, and the replacement tree of <code>EntityReference</code> nodes would be updated. Other actions might take place as well, as specified by the <code>domConfig</code> property.
<code>Node renameNode(Node n, DOMString namespaceURI, DOMString qualifiedName)</code>	Renames the node, specified by <code>n</code> , using the new namespace URI and qualified name

DOMImplementation

The `DOMImplementation` interface provides methods that are not specific to any particular document, but to any document from this DOM implementation. You can get a `DOMImplementation` object from the `implementation` property of the `Document` interface.

Method	Description
<code>boolean hasFeature(DOMString feature, DOMString version)</code>	Returns a boolean indicating whether this DOM implementation supports the <code>feature</code> requested. <code>version</code> is the version number of the feature to test.
<code>DocumentType createDocumentType(DOMString qualifiedName, DOMString publicID, DOMString systemID)</code>	Creates a <code>DocumentType</code> object with the specified attributes
<code>Document createDocument(DOMString namespaceURI, DOMString qualifiedName, DocumentType doctype)</code>	Creates a <code>Document</code> object, with the document element specified by <code>qualifiedName</code> . The <code>doctype</code> property must refer to an object of type <code>DocumentType</code> .
<code>DOMObject getFeature(DOMString feature, DOMString version)</code>	Returns an object that implements the required APIs specified in the <code>feature</code> and <code>version</code> attributes

DOMImplementationSource

This interface is used to get a DOM implementation. It is also used to get the `DOMImplementationList` (in the next table).

Appendix D: The XML Document Object Model

Method	Description
<code>DOMImplementation</code> <code>getDOMImplementation</code> (DOMString features)	Returns a <code>DOMImplementation</code> that supports the features specified in the <code>features</code> attribute
<code>DOMImplementationList</code> <code>getDOMImplementationList</code> (DOMString features)	Returns a <code>DOMImplementationList</code> of DOM implementations that support the features specified in the <code>features</code> attribute. Feature names are not case sensitive, meaning that “Core” would be treated the same as “core.”

Keep in mind that the list of “features” a DOM implementation might support are limitless; anyone implementing the DOM recommendation could think of other enhanced functionality they might want to provide, and create a feature. *Every* DOM implementation supports the “core” feature, because that feature represents the core interfaces. DOM implementations implementing the “extended interfaces” also support the “XML” feature. DOM implementations that support the Load and Save functionality support the “LS” feature, while DOM implementations that support the Validation interfaces will support the “Validation” feature.

DOMImplementationList

There are several different, interrelated modules in relation to the DOM. For example, this appendix lists the Core, Load and Save, and Validation modules, but many others might also be implemented. Furthermore, there have been various stages in the evolution of the DOM recommendations. This appendix covers DOM Level 3, which is the third incarnation of the DOM recommendations, while previous editions of this book covered DOM Level 2.

Therefore, a toolkit might provide more than one DOM implementation that developers could use; one might implement DOM Level 2, while another implements various modules from DOM Level 3. The `DOMImplementationList` interface is used to get the various DOM implementations provided by a particular toolkit.

Property	Type	Description
<code>length</code>	unsigned long (read-only)	Returns the number of items in the list

Method	Description
<code>DOMImplementation</code> item (unsigned long index)	Returns the <code>DOMImplementation</code> at the position indicated by the <code>index</code> parameter

The items are numbered starting at 0, so the first item is “item 0,” the second is “item 1,” etc.

DocumentFragment

A document fragment is a temporary holding place for a group of nodes, usually created with the intent of inserting the nodes back into the document at a later point. The `DocumentFragment` interface extends the `Node` interface, without adding any additional properties or methods.

NodeList

A `NodeList` contains an ordered group of nodes, accessed via an index.

Property	Type	Description
<code>length</code>	unsigned long (read-only)	The number of nodes contained in this list. The range of valid child node indices is 0 to <code>length - 1</code> inclusive.

Method	Description
<code>Node item(unsigned long index)</code>	Returns the <code>Node</code> in the list at the indicated index. If <code>index</code> is the same as or greater than <code>length</code> , it returns <code>NULL</code> .

DOMStringList

The `DOMStringList` interface is used for working with a collection of strings. It represents an *array* or *collection* of `DOMString` objects.

Property	Type	Description
<code>length</code>	unsigned long (read-only)	The number of <code>DOMString</code> objects in this list

Method	Description
<code>boolean contains(DOMString str)</code>	Indicates whether the string in the <code>str</code> parameter is in this list of <code>DOMString</code> objects. That is, if one of the <code>DOMString</code> objects in the list is the same as the <code>str</code> parameter, <code>contains()</code> will return <code>true</code> ; otherwise, it will return <code>false</code> .
<code>DOMString item(unsigned long index)</code>	Returns the <code>DOMString</code> from this list, which is at the location specified by the <code>index</code> parameter. Returns <code>NULL</code> if there is no item with the specified index location.

The items are numbered starting at 0, so the first item is "item 0," the second is "item 1," etc.

NameList

The `NameList` interface is similar to the `DOMStringList` interface, except that it is used for an *array* or *collection* of namespace names and values, instead of `DOMString` objects.

Property	Type	Description
<code>length</code>	unsigned long (read-only)	Returns the number of items in the list

Method	Description
<code>DOMString getName(unsigned long index)</code>	Returns the namespace name at the position indicated by the <code>index</code> parameter
<code>DOMString getNamespaceURI(unsigned long index)</code>	Returns the namespace URI for the item at the position indicated by the <code>index</code> parameter
<code>boolean contains(DOMString str)</code>	Returns <code>true</code> if the specified name is part of this <code>NameList</code> ; otherwise, it returns <code>false</code>
<code>boolean containsNS(DOMString namespaceURI, DOMString name)</code>	Returns <code>true</code> if the namespace URI/name combination exists in this <code>NameList</code> ; otherwise, it returns <code>false</code>

The items are numbered starting at 0, so the first item is “item 0,” the second is “item 1,” etc.

Element

The `Element` interface provides properties and methods for working with an element. It extends the `Node` interface.

Property	Type	Description
<code>tagName</code>	<code>DOMString</code> (read-only)	The name of the element
<code>schemaTypeInfo</code>	<code>TypeInfo</code> (read-only)	The type information associated with this element. See the <code>TypeInfo</code> interface for more information.

Method	Description
<code>DOMString getAttribute(DOMString name)</code>	Returns the value of the attribute with the specified <code>name</code> , or an empty string if that attribute does not have a specified or default value
<code>void setAttribute(DOMString name, DOMString value)</code>	Sets the value of the specified attribute to this new <code>value</code> . If no such attribute exists, a new one with this <code>name</code> is created.

Appendix D: The XML Document Object Model

Method	Description
<code>void removeAttribute</code> (DOMString name)	Removes the specified attribute. If the attribute has a default value, it is immediately replaced with an identical attribute containing this default value.
<code>Attr getAttributeNode</code> (DOMString name)	Returns an <code>Attr</code> node containing the named attribute. Returns <code>NULL</code> if there is no such attribute.
<code>Attr setAttributeNode</code> (Attr newAttr)	Adds a new attribute node. If an attribute with the same name already exists, it is replaced. If an <code>Attr</code> has been replaced, it is returned; otherwise, <code>NULL</code> is returned.
<code>Attr removeAttributeNode</code> (Attr oldAttr)	Removes the specified <code>Attr</code> node and returns it. If the attribute has a default value, it is immediately replaced with an identical attribute containing this default value.
<code>NodeList getElementsByTagName</code> (DOMString name)	Returns a <code>NodeList</code> of all descendants with the given node name
<code>DOMString getAttributeNS</code> (DOMString namespaceURI, DOMString localName)	Returns the value of the specified attribute, or an empty string if that attribute does not have a specified or default value
<code>void setAttributeNS</code> (DOMString namespaceURI, DOMString qualifiedName, DOMString value)	Sets the value of the specified attribute to this new value. If no such attribute exists, a new one with this namespace URI and QName is created.
<code>void removeAttributeNS</code> (DOMString namespaceURI, DOMString localName)	Removes the specified attribute. If the attribute has a default value, it is immediately replaced with an identical attribute containing this default value.
<code>Attr getAttributeNodeNS</code> (DOMString namespaceURI, DOMString localName)	Returns an <code>Attr</code> node containing the specified attribute. Returns <code>NULL</code> if there is no such attribute.
<code>Attr setAttributeNodeNS</code> (Attr newAttr)	Adds a new <code>Attr</code> node to the list. If an attribute with the same namespace URI and local name exists, it is replaced. If an <code>Attr</code> object is replaced, it is returned; otherwise, <code>NULL</code> is returned.
<code>NodeList getElementsByTagNameNS</code> DOMString namespaceURI, DOMString localName)	Returns a <code>NodeList</code> of all the elements matching these criteria
<code>boolean hasAttribute</code> (DOMString name)	Returns <code>true</code> when the node has an attribute with the given name — whether explicitly or by default from the schema
<code>boolean hasAttributeNS</code> (DOMString namespaceURI, DOMString localName)	Returns <code>true</code> when the node has an attribute with the given namespace URI and local name — whether explicitly or by default from the schema

Table continued on following page

Appendix D: The XML Document Object Model

Method	Description
<code>setIdAttribute(DOMString name, boolean isId)</code>	Used to make an attribute an ID attribute or to set it to <i>not</i> be an ID attribute; the <code>isId</code> parameter specifies whether the attribute specified by <code>name</code> should be set to an ID or not.
<code>setIdAttributeNS(DOMString namespaceURI, DOMString localName, boolean isId)</code>	Used to make an attribute an ID attribute or to set it to <i>not</i> be an ID attribute; the <code>isId</code> parameter specifies whether the attribute specified by the namespace URI and local name should be set to an ID or not.
<code>setIdAttributeNode(Attr idAttr, boolean isId)</code>	Sets the specified attribute node to be an ID or not to be an ID; the <code>isId</code> parameter specifies whether the node should be set to an ID or not.

NamedNodeMap

A named node map represents an unordered collection of nodes, retrieved by name.

Property	Type	Description
<code>length</code>	unsigned long (read-only)	The number of nodes in the map

Method	Description
Node <code>getNamedItem(DOMString name)</code>	Returns a <code>Node</code> , where the <code>nodeName</code> is the same as the <code>name</code> specified, or <code>NULL</code> if no such node exists
Node <code>setNamedItem(Node arg)</code>	The <code>arg</code> parameter is a <code>Node</code> object, which is added to the list. The <code>nodeName</code> property is used for the name of the node in this map. If a node with the same name already exists, it is replaced. If a <code>Node</code> is replaced, it is returned; otherwise, <code>NULL</code> is returned.
Node <code>removeNamedItem(DOMString name)</code>	Removes the <code>Node</code> specified by <code>name</code> and returns it
Node <code>item(unsigned long index)</code>	Returns the <code>Node</code> at the specified <code>index</code> . If <code>index</code> is the same as or greater than <code>length</code> , it returns <code>NULL</code> .
Node <code>getNamedItemNS(DOMString namespaceURI, DOMString localName)</code>	Returns a <code>Node</code> matching the namespace URI and local name, or <code>NULL</code> if no such node exists
Node <code>setNamedItemNS(Node arg)</code>	The <code>arg</code> parameter is a <code>Node</code> object, which is added to the list. If a node with the same namespace URI and local name already exists, it is replaced. If a <code>Node</code> is replaced, it is returned; otherwise, <code>NULL</code> is returned.
Node <code>removeNamedItemNS(DOMString namespaceURI, DOMString localName)</code>	Removes the specified node and returns it

Attr

The `Attr` interface provides properties for dealing with an attribute. It extends the `Node` interface.

Property	Type	Description
<code>name</code>	<code>DOMString</code> (read-only)	The name of the attribute
<code>specified</code>	boolean (read-only)	A boolean indicating whether this attribute was specified (<code>true</code>) or just defaulted (<code>false</code>)
<code>value</code>	<code>DOMString</code>	The value of the attribute
<code>ownerElement</code>	<code>Element</code> (read-only)	An <code>Element</code> object, representing the element to which this attribute belongs
<code>schemaTypeInfo</code>	<code>TypeInfo</code> (read-only)	The type information associated with this attribute by its schema. See the <code>TypeInfo</code> interface for more information.
<code>isId</code>	boolean (read-only)	Returns <code>true</code> if this attribute is an ID attribute, meaning that it was specified to be such by the document's DTD or schema.

CharacterData

The `CharacterData` interface provides properties and methods for working with character data. It extends the `Node` interface.

Property	Type	Description
<code>data</code>	<code>DOMString</code>	The text in this <code>CharacterData</code> node
<code>length</code>	unsigned long (read-only)	The number of characters in the node

Method	Description
<code>DOMString substringData(unsigned long offset, unsigned long count)</code>	Returns a portion of the string, starting at the <code>offset</code> . Will return the number of characters specified in <code>count</code> or until the end of the string, whichever is less.
<code>void appendData(DOMString arg)</code>	Appends the string in <code>arg</code> to the end of the string
<code>void insertData(unsigned long offset, DOMString arg)</code>	Inserts the string in <code>arg</code> into the middle of the string, starting at the position indicated by <code>offset</code>
<code>void deleteData(unsigned long offset, unsigned long count)</code>	Deletes a portion of the string, starting at the <code>offset</code> . Will delete the number of characters specified in <code>count</code> or until the end of the string, whichever is less.

Table continued on following page

Appendix D: The XML Document Object Model

Method	Description
<code>void replaceData(unsigned long offset, unsigned long count, DOMString arg)</code>	Replaces a portion of the string, starting at the <code>offset</code> . Will replace the number of characters specified in <code>count</code> or until the end of the string, whichever is less. The <code>arg</code> parameter is the new string to be inserted.

Text

The `Text` interface provides additional methods and properties for working with text nodes. It extends the `CharacterData` interface.

Property	Type	Description
<code>isElementContentWhitespace</code>	boolean (read-only)	Returns <code>true</code> if the text content is “element content whitespace” — the type of whitespace that can usually be ignored by a parser, because the element is not declared in its DTD or schema to have text content — or <code>false</code> otherwise
<code>wholeText</code>	DOMString (read-only)	Returns a <code>DOMString</code> containing all of the <code>Text</code> nodes logically adjacent to this one. In other words, it would be similar to normalizing the parent node of this <code>Text</code> node, and then getting the text.

Method	Description
<code>Text splitText(unsigned long offset)</code>	Separates this single <code>Text</code> node into two adjacent <code>Text</code> nodes. All of the text up to the <code>offset</code> point goes into the first <code>Text</code> node, and all of the text starting at the <code>offset</code> point to the end goes into the second <code>Text</code> node.
<code>Text replaceWholeText(DOMString content)</code>	Replaces the text of this node — <i>and all logically adjacent Text nodes</i> — with the text specified in the <code>DOMString</code> parameter. In other words, if you have previously used the <code>splitText()</code> method to split this <code>Text</code> node into multiple <code>Text</code> nodes, then calling <code>replaceWholeText()</code> would remove all of those <code>Text</code> nodes and replace them with one single node containing the contents of the <code>content</code> parameter.

Comment

The `Comment` interface encapsulates an XML comment. It extends the `CharacterData` interface, without adding any additional properties or methods. As mentioned in Chapter 2, however, remember that not all XML parsers will pass on comments, so your DOM implementation can only make comments available to you if the parser it’s using under the hood gives it access to them.

TypeInfo

The `TypeInfo` interface represents type information, as specified by a document's DTD or schema.

Property	Type	Description
<code>typeName</code>	<code>DOMString</code> (read-only)	The name of the attribute's or element's type, or <code>NULL</code> if unknown
<code>type Namespace</code>	<code>DOMString</code> (read-only)	The namespace of the attribute's or element's type, or <code>NULL</code> if unknown.

If the node in question is an attribute, and the schema for the document is a DTD, the namespace returned is `http://www.w3.org/TR/REC-xml`.

DerivationMethods Constant	Description
<code>DERIVATION_RESTRICTION</code>	Indicates derivation by restriction
<code>DERIVATION_EXTENSION</code>	Indicates derivation by extension
<code>DERIVATION_UNION</code>	Indicates derivation by union
<code>DERIVATION_LIST</code>	Indicates derivation by list

Method	Description
<code>boolean isDerivedFrom (DOMString typeNamespaceArg, DOMString typeNameArg, unsigned long derivationMethod)</code>	Returns <code>true</code> if the type of the current node is derived from the specified type

UserDataHandler

If you're going to make use of "user data," you should implement the `UserDataHandler` interface in your code. When user data is specified for a node and that node is cloned, renamed, or imported, the DOM implementation will call your object, implementing this interface, enabling you to do whatever you wish with the data.

OperationType Constant	Description
<code>NODE_CLONED</code>	The node is being cloned.
<code>NODE_IMPORTED</code>	The node is being imported.
<code>NODE_DELETED</code>	The node is being deleted.
<code>NODE_RENAMED</code>	The node is being renamed.
<code>NODE_ADOPTED</code>	The node is being adopted by a new parent node.

Appendix D: The XML Document Object Model

Method	Description
<code>void handle(unsigned short operation, DOMString key, DOMUserData data, Node src, Node dest)</code>	Whenever a node for which this handler is registered is cloned, imported, deleted, renamed, or adopted, this method is called. The <code>operation</code> parameter is one of the <code>OperationType</code> values defined above.

DOMLocator

This interface describes a location in an XML document. For example, it might be used to indicate where an error occurred.

Property	Type	Description
<code>line Number</code>	long (read-only)	The line number this locator is pointing to, or -1 if not available
<code>column Number</code>	long (read-only)	The column number this locator is pointing to, or -1 if not available
<code>byteOffset</code>	long (read-only)	The number of bytes into the document this locator is pointing to, or -1 if no byte offset is available
<code>utf16Offset</code>	long (read-only)	The UTF 16 offset into the document this locator is pointing to, or -1 if there is no UTF 16 offset available
<code>related Node</code>	Node (read-only)	The node this locator is pointing to, or <code>NULL</code> if no node is available
<code>uri</code>	DOMString	The URI this locator is pointing to, or <code>NULL</code> if no URI is available

DOMConfiguration

This interface represents a document's configuration and parameters. Using this interface, developers can change many aspects of the way a DOM behaves.

The list of parameters that can be used with `DOMConfiguration` is limitless. Several are defined in the DOM Recommendation(s), but others can be created by other recommendations or specifications or even by specific DOM implementers.

Property	Type	Description
<code>parameter Names</code>	DOMStringList (read-only)	The list of parameters supported by this <code>DOMConfiguration</code> object — in other words, by this DOM implementation

Method	Description
<code>setParameter(DOMString name, DOMUserData value)</code>	Sets the value of the parameter
<code>DOMUserData getParameter(DOMString name)</code>	Returns the value of the parameter, if known
<code>boolean canSetParameter(DOMString name, DOMUserData value)</code>	Returns <code>true</code> if the DOM implementation is able to set the given parameter to the given value

Extended Interfaces: XML Module

So far, we've been looking at the Core DOM interfaces; these are interfaces that must always be implemented, by every DOM implementation. However, not every DOM implementation is meant for working with XML documents; some DOM implementations only work with HTML documents.

The XML Module provides the DOM Extended Interfaces for XML, which need only be provided by DOM implementations that will be working with XML documents.

CDATASection

The `CDATASection` interface encapsulates an XML CDATA section. It extends the `Text` interface, without adding any additional properties or methods.

ProcessingInstruction

The `ProcessingInstruction` interface provides properties for working with an XML processing instruction (PI). It extends the `Node` interface.

Property	Type	Description
<code>target</code>	<code>DOMString</code> (read-only)	The PI target — in other words, the name of the application to which the PI should be passed
<code>data</code>	<code>DOMString</code>	The content of the PI

DocumentType

The `DocumentType` interface provides properties for working with an XML document type. It can be retrieved from the `Document` interface's `doctype` property. (If a document doesn't have a document type, `doctype` will return `NULL`.) `DocumentType` extends the `Node` interface.

Appendix D: The XML Document Object Model

Property	Type	Description
name	DOMString (read-only)	The name of the DTD
entities	NamedNodeMap (read-only)	A <code>NamedNodeMap</code> containing all entities declared in the DTD (both internal and external). Parameter entities are not contained, and duplicates are discarded according to the rules followed by validating XML parsers.
notations	NamedNodeMap (read-only)	A <code>NamedNodeMap</code> containing the notations contained in the DTD. Duplicates are discarded.
publicId	DOMString (read-only)	The external subset's public identifier
systemId	DOMString (read-only)	The external subset's system identifier
internalSubset	DOMString (read-only)	The internal subset, as a string

Notation

The `Notation` interface provides properties for working with an XML notation. Notations are read-only in the DOM. It extends the `Node` interface.

Property	Type	Description
publicId	DOMString (read-only)	The public identifier of this notation. If the public identifier was not specified, it returns <code>NULL</code> .
systemId	DOMString (read-only)	The system identifier of this notation. If the system identifier was not specified, it returns <code>NULL</code> .

Entity

The `Entity` interface provides properties for working with parsed and unparsed entities. `Entity` nodes are read-only. This interface extends the `Node` interface.

Property	Type	Description
publicId	DOMString (read-only)	The public identifier associated with the entity, or <code>NULL</code> if none is specified
systemId	DOMString (read-only)	The system identifier associated with the entity, or <code>NULL</code> if none is specified
notationName	DOMString (read-only)	For unparsed entities, the name of the notation for the entity. <code>NULL</code> for parsed entities.

Property	Type	Description
input Encoding	DOMString (read-only)	The encoding used for this entity at the time of parsing, <i>when it is an external parsed entity</i> . (It is <code>NULL</code> otherwise.)
xml Encoding	DOMString (read-only)	The encoding of this entity, <i>when it is an external parsed entity</i> . (It is <code>NULL</code> otherwise.)
xmlVersion	DOMString (read-only)	The XML version number of the entity, <i>when it is an external parsed entity</i> . (It is <code>NULL</code> otherwise.)

EntityReference

The `EntityReference` interface encapsulates an XML entity reference. It extends the `Node` interface, without adding any properties or methods.

Load and Save Interfaces

The interfaces defined in this section are used for loading and saving XML documents. These interfaces are a welcome addition to the DOM Recommendations; when the DOM Level 2 Recommendation(s) were published, there was no standard way to load an XML document into a DOM implementation nor to save it.

You will notice that the interface and data type names all contain “LS” — short for “Load and Save” — to distinguish them from other interface names.

Data Types

In addition to the data types listed earlier in this appendix — `DOMString`, `DOMObject`, etc. — some fundamental data types are defined specifically for the Load and Save interfaces.

First, there are two data types defined for a sequence of bytes, representing data. `LSInputStream` defines a sequence of bytes *into* a DOM, while `LSOutputStream` defines a sequence of bytes *out of* a DOM. For reference, the `LSInputStream` would be analogous to the Java `java.io.InputStream` object, and the `LSOutputStream` would be analogous to the Java `java.io.OutputStream` object.

While these two stream data types are used for working with a series of bytes, there are also two data types defined for working with 16-bit units, such as UTF-16 characters. These are `LSReader` and `LSWriter`.

LSException

This interface defines an exception that can be raised when reading or writing a document with a DOM implementation.

Appendix D: The XML Document Object Model

Property	Type	Description
code	unsigned short	The exception code, which will be one of the values in the following table

LSExceptionCode Constant	Description
PARSE_ERR	The error was a result of parsing an XML document.
SERIALIZE_ERR	The error was a result of writing an XML document.

DOMImplementationLS

This interface contains factory methods for creating load and save objects.

Note that not all DOM implementations support asynchronous processing. To determine whether your DOM implementation does, use the `DOMImplementation.getFeature()` method to find out if the DOM implementation supports the LS-Async feature.

DOMImplementationLSMode Constant	Description
MODE_SYNCHRONOUS	Synchronous mode — the method will not return until the document is finished loading or writing
MODE_ASYNCHRONOUS	Asynchronous mode — the method will return immediately, and processing will continue in the background. When working asynchronously, objects in your application will have to implement the <code>LSLoadEvent</code> or <code>LSProgressEvent</code> interface, so that the parser can inform the application when parsing is complete.

Method	Description
<code>LSParser createLSParser(unsigned short mode, DOMString schemaType)</code>	Creates a new <code>LSParser</code> object, for use in parsing a document
<code>LSSerializer createLSSerializer()</code>	Creates a new <code>LSSerializer</code> object, for use in serializing a document
<code>LSInput createLSInput()</code>	Creates a new, empty <code>LSInput</code> object
<code>LSOutput createLSOutput()</code>	Creates a new, empty <code>LSOutput</code> object

LSParser

This interface is used to parse an XML document from scratch or to augment an already existing XML document.

Appendix D: The XML Document Object Model

Property	Type	Description
domConfig	DOMConfiguration (read-only)	The <code>DOMConfiguration</code> object, which will be used when parsing an input source. This object can be used to configure how the input will be parsed.
filter	LSParserFilter	When a filter is provided, it can cause the result of the parse operation to omit anything that was filtered. For more information, see the <code>LSParseFilter</code> interface.
async	boolean (read-only)	Returns <code>true</code> if the <code>LSParser</code> is asynchronous, <code>false</code> otherwise
busy	boolean (read-only)	Returns <code>true</code> if the <code>LSParser</code> is currently loading a document, <code>false</code> otherwise

ActionType Constant	Description
<code>ACTION_APPEND_AS_CHILDREN</code>	The result of the parsing should be appended as children of the context node.
<code>ACTION_REPLACE_CHILDREN</code>	All children of the context node should be replaced by the result of the parsing.
<code>ACTION_INSERT_BEFORE</code>	The result of the parse operation should be inserted into the document as children of the context node, before any other children.
<code>ACTION_INSERT_AFTER</code>	The result of the parse operation should be inserted into the document as children of the context node, after any other children.
<code>ACTION_REPLACE</code>	The context node should be replaced by the result of the parse operation.

Method	Description
<code>Document parse(LSInput input)</code>	Parse an XML document from the <code>LSInput</code> input source.
<code>Document parseURI(DOMString uri)</code>	Parse an XML document that resides at the specified URI.
<code>Node parseWithContext(LSInput input, Node contextArg, unsigned short action)</code>	Parse an XML document from the <code>LSInput</code> input source, and put the result into the context of the <code>contextArg</code> node. The <code>action</code> parameter would specify the type of action, per the <code>ActionType</code> constants listed above.
<code>abort()</code>	Aborts the current parse process. If nothing is currently being parsed, this method does nothing.

LSSerializer

This interface represents an object that can serialize an XML document to an output stream or a string.

Property	Type	Description
domConfig	DOMConfig (read-only)	The <code>DOMConfiguration</code> object used to configure how the document will be serialized
newLine	DOMString	The end-of-line sequence characters to be used in the XML being written out
filter	LSSerializerFilter	If provided, the filter can be used to control which parts of the XML document will be serialized.

Method	Description
boolean write(Node nodeArg, LSOutput destination)	Serializes the XML to the <code>destination</code> parameter. Returns <code>true</code> if successful, <code>false</code> otherwise.
boolean writeToURI(Node nodeArg, DOMString uri)	A convenience method; it is the same as calling <code>write()</code> with the <code>destination</code> specifying no encoding, and the <code>systemId</code> set to a URI.
DOMString writeToString(Node nodeArg)	Serializes the XML, and returns it in the <code>DOMString</code> return value

LSInput

This interface is used for working with the “input” to a parse operation. It doesn’t provide any methods, just attributes, which specify the source XML document.

Property	Type	Description
characterStream	LSReader	A stream of characters that contains the XML to be parsed
byteStream	LSInputStream	A stream of bytes that contains the XML to be parsed
stringData	DOMString	A <code>DOMString</code> that contains the XML to be parsed. If you have a string containing XML data, this is the property you want to use, not the <code>characterStream</code> property.
systemId	DOMString	The system ID for the input XML
publicId	DOMString	The public identifier of the input XML
baseURI	DOMString	The base URI to be used when resolving a relative system ID

Property	Type	Description
encoding	DOMString	The encoding of the XML, if known
certifiedText	boolean	Set this attribute to <code>true</code> if you want the parser to assume that the XML has been “certified,” as specified in section 2.13 of XML 1.1.

LSOutput

This interface represents an output destination, where XML data will be serialized.

Property	Type	Description
characterStream	LSWriter	A writeable stream of 16-bit characters, where the serialized XML will be sent
byteStream	LSOutputStream	A writeable stream of bytes, where the serialized XML will be sent
systemId	DOMString	A URI for the output destination
encoding	DOMString	The character encoding to use for the serialized XML

LSResourceResolver

This interface is used for resolving external resources. It defines only one method.

Method	Description
LSInput resolveResource(DOMString type, DOMString namespaceURI, DOMString publicId, DOMString systemId, DOMString baseURI)	Resolves the external resource, specified by the various input parameters. The result of the resolution is returned in an <code>LSInput</code> object.

LSParserFilter

This interface can be used during parsing, to filter the result. For example, you might want to parse an XML document but specifically ignore some sections of it that you know you don’t need.

Property	Type	Description
whatToShow	unsigned long (read-only)	Indicates to the <code>LSParser</code> what types of nodes to show to the <code>acceptNode()</code> method

Appendix D: The XML Document Object Model

Filter Constant	Description
FILTER_ACCEPT	Accept the node.
FILTER_REJECT	Reject the node and its children.
FILTER_SKIP	Skip this single node; children will still be looked at.
FILTER_INTERRUPT	Interrupt the normal processing of the document.

Method	Description
unsigned short startElement (Element elementArg)	<p>This method is called at the beginning of the parsing of each element, by the parser. The return value indicates to the parser how it should handle the node, using one of the constants defined in the preceding table.</p> <p>This method is called <i>before</i> the element is parsed, not at the end, so it can be used to quickly skip processing of an element.</p>
unsigned short acceptNode (Node nodeArg)	<p>This method is called by the parser at the completion of parsing each node. It returns to the parser how the node should be handled, using one of the constants defined in the preceding table.</p>

LSSerializerFilter

Similar to the `LSParserFilter`, this interface can be used to filter an XML document that is being serialized and to control which parts are written to the output.

Property	Type	Description
whatToShow	unsigned long (read-only)	Indicates the type of nodes to show the filter

LSProgressEvent

This event is raised at various points during the parsing of a document, to indicate progress. It has no methods, just attributes, which provide information about the progress.

Property	Type	Description
input	LSInput (read-only)	The input source that is currently being parsed
position	unsigned long (read-only)	The current position of the parser
totalSize	unsigned long (read-only)	The total size of the document being parsed — including all external entities

LSLoadEvent

This interface defines an event that is raised when a document has completed loading. As with the `LSProgressEvent` interface, it has no methods, just properties.

Property	Type	Description
new Document	Document (read-only)	The document that finished loading
input	LSInput (read-only)	The input source that was parsed

Validation Interfaces

The interfaces in this section are concerned with document validation — that is, ensuring that a document conforms to a DTD or schema.

For the purposes of the DOM Validation interfaces, no particular schema technology is implied; when the word “schema” is referred to, it may be the W3C XML Schema Recommendation, but the DOM interfaces don’t assume this. Any other schema technology could be used, as long as the DOM implementation you’re using supports it.

As a naming convention, all of the interfaces in the DOM Validation module end with “VAL.”

ExceptionVAL

There is just one exception interface for the DOM Validation functionality, which is the `ExceptionVAL` exception.

Property	Type	Description
code	unsigned short	A code indicating the reason for the exception. This code is one of the values in the <code>ExceptionVALCode</code> constants — of which there is only one!

Appendix D: The XML Document Object Model

ExceptionVALCode Constant	Description
NO_SCHEMA_AVAILABLE_ERR	Indicates that the operation could not be performed, because the schema was not available

NodeEditVAL

This interface is used for validation of a particular node in a document. The majority of methods defined for this interface are simply questions the programmer can ask: Is it okay if I do this or will it make the document invalid?

Property	Type	Description
defaultValue	DOMString (read-only)	The element's or attribute's default value, according to the schema, if any
enumerated Values	DOMStringList (read-only)	If the element or attribute is defined in its schema to have an enumerated list of values, this property will return the list of possible values.

Validation Type Constant	Description
VAL_WF	Check whether the node is well-formed
VAL_NS_WF	Check whether the node is namespace well-formed
VAL_INCOMPLETE	This type of validation checks only the node's immediate children. This type of validation also includes VAL_NS_WF.
VAL_SCHEMA	Check whether the node, and all of its descendants, are valid according to the document's schema

Validation State Constant	Description
VAL_TRUE	The node is valid according to the operation performed.
VAL_FALSE	The node is not valid according to the operation performed.
VAL_UNKNOWN	The node's validity is unknown.

Method	Description
unsigned short canInsertBefore (Node newChild)	Indicates whether the specified node could be inserted before this one, according to the schema
unsigned short canRemoveChild (Node oldChild)	Indicates whether the specified node could be removed from the document, according to the schema

Method	Description
unsigned short canReplaceChild (Node newChild, Node oldChild)	Indicates whether the specified new node could be used to replace the old one, according to the schema
unsigned short canAppendChild (Node newChild)	Indicates whether the specified node could be appended to this one, according to the schema
unsigned short nodeValidity (unsigned short valType)	Returns the current validity of the node

All of these methods return a Validation State constant, indicating the validity being requested.

DocumentEditVAL

This interface extends the `NodeEditVAL` interface, and is used for validating an entire document. The Recommendation states that an object which implements this interface must also implement the `Document` interface.

Property	Type	Description
continuous Validity Checking	boolean	Setting this property to <code>true</code> indicates that the programmer wants the DOM implementation to continually check the document for validity; that is, every time an element or attribute is added, removed, or modified, the DOM implementation should ensure that the document is still valid.
domConfig	DOMConfiguration (read-only)	The <code>DOMConfiguration</code> object, which can be used for validation-related settings. Note that this is redundant, as the Recommendation states that objects implementing <code>DocumentEditVAL</code> must also implement <code>Document</code> , which has its own <code>domConfig</code> attribute!

Method	Description
NameList getDefinedElements (DOMString namespaceURI)	Returns a list of all of the element names defined (with global declaration) for the specified namespace. If no schema is available (or there are no names for the specified namespace), <code>NULL</code> is returned.
unsigned short validateDocument()	Validates the document against its schema. The result of the validation is returned; see the <code>NodeEditVAL</code> interface for information on the Validation State constants.

ElementEditVAL

This interface extends the `NodeEditVAL` interface for functionality specifically related to the validation of elements.

Appendix D: The XML Document Object Model

Most of the properties simply return lists, indicating what the document's schema will permit the document to contain; for example, the `allowedChildren` property returns a `NameList`, which contains the names of all elements that are allowed as children of this element, per the schema.

Many of these properties and methods would be more useful when creating a document than when working with a document that's already been fully parsed. Or, if validation has been turned off, the properties and methods here can be used to programmatically ensure that the document adheres to its schema.

Property	Type	Description
<code>allowedChildren</code>	<code>NameList</code> (read-only)	A list of all child elements that can be children of this element, including wildcards
<code>allowedFirstChilden</code>	<code>NameList</code> (read-only)	A list of all child elements that could appear as the first child of this element
<code>allowedParents</code>	<code>NameList</code> (read-only)	A list of all elements that could possibly be the parent of this element
<code>allowedNextSiblings</code>	<code>NameList</code> (read-only)	A list of all elements that could follow this element in the document
<code>allowedPrevious Siblings</code>	<code>NameList</code> (read-only)	A list of all elements that could precede this element in the document
<code>allowedAttributes</code>	<code>NameList</code> (read-only)	A list of attribute names that can be attached to this element
<code>requiredAttributes</code>	<code>NameList</code> (read-only)	A list of all attributes that <i>must</i> appear on this element
<code>contentType</code>	unsigned short (read-only)	The element's content type, as defined in the <code>ContentTypeVAL</code> constants (see the next table)

<code>ContentTypeVAL</code> Constant	Description
<code>VAL_EMPTY_CONTENTTYPE</code>	The element has no content
<code>VAL_ANY_CONTENTTYPE</code>	The element contains unordered children — corresponds to the ANY content model used by DTDs
<code>VAL_MIXED_CONTENTTYPE</code>	The element can have child elements, along with text children
<code>VAL_ELEMENTS_CONTENTTYPE</code>	The element contains only elements (optionally with whitespace)
<code>VAL_SIMPLE_CONTENTTYPE</code>	The element has only text content

Method	Description
unsigned short canSetTextContent (DOMString possibleTextContent)	Indicates whether the specified text could be set as this element's text content
unsigned short canSetAttribute (DOMString attrname, DOMString attrval)	Indicates whether the specified attribute could be set on this element
unsigned short canSetAttributeNode (Attr attrNode)	Indicates whether the specified attribute could be set on this element
unsigned short canSetAttributeNS (DOMString namespaceURI, DOMString qualifiedName, DOMString value)	Indicates whether the specified attribute could be set on this element
unsigned short canRemoveAttribute (DOMString attrname)	Indicates whether the specified attribute could be removed from the element
unsigned short canRemoveAttributeNS (DOMString namespaceURI, DOMString localName)	Indicates whether the specified attribute could be removed from the element
unsigned short canRemoveAttributeNode (Node attrNode)	Indicates whether the specified attribute could be removed from the element
unsigned short isElementDefined (DOMString name)	Determines whether this element is actually defined in the document's schema
unsigned short isElementDefinedNS (DOMString namespaceURI, DOMString name)	Determines whether this element is actually defined in the document's schema

CharacterDataEditVAL

This interface extends the `NodeEditVAL` interface. It is for working with character data within a document. No attributes are defined aside from those already defined for the `NodeEditVAL` interface, just methods that can be used to determine validity under various conditions.

Method	Description
unsigned short isWhitespaceOnly()	Indicates whether the character data is defined in the schema to be whitespace
unsigned short canSetData (DOMString arg)	Indicates whether the specified text can be set as this character data's content
unsigned short canAppendData (DOMString arg)	Indicates whether the specified text can be appended to this character data's content

Table continued on following page

Appendix D: The XML Document Object Model

Method	Description
unsigned short canReplaceData (unsigned long offset, unsigned long count, DOMString arg)	Indicates whether the specified text can be used to replace a section of this character data's content
unsigned short canInsertData (unsigned long offset, DOMString arg)	Indicates whether the specified text can be inserted into this character data's content at the specified location
unsigned short canDeleteData(unsigned long offset, unsigned long count)	Indicates whether the character data content can be deleted