Document Object Model (DOM) Level 3 Content Models and Load and Save Specification

Version 1.0

W3C Working Draft *19 April 2001*

This version:
http://www.w3.org/TR/2001/WD-DOM-Level-3-CMLS-20010419
(PostScript file , PDF file , plain text , ZIP file , single HTML file)
Latest version:
http://www.w3.org/TR/DOM-Level-3-CMLS
Previous version:
http://www.w3.org/TR/2001/WD-DOM-Level-3-CMLS-20010209

Editors:
Ben Chang, *Oracle*
Andy Heninger, *IBM*
Joe Kesselman, *IBM*
Rezaur Rahman, *Intel Corporation*

Abstract

This specification defines the Document Object Model Content Models and Load and Save Level 3, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model Content Models and Load and Save Level 3 builds on the Document Object Model Core Level 3.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

Comments on this document are invited and are to be sent to the public mailing list www-dom@w3.org. An archive is available at http://lists.w3.org/Archives/Public/www-dom/.

This document has been produced as part of the W3C DOM Activity. The authors of this document are the DOM WG members.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Table of contents

Expanded Table of Contents

Expanded Table of Contents

Copyright Notice

W3C Document Copyright Notice and License

No right to create modifications or derivatives of W3C documents is granted pursuant to this license. However, if additional requirements (documented in the Copyright FAQ) are satisfied, the right to create modifications or derivatives is sometimes granted by the W3C to individuals complying with those requirements.

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

W3C Software Copyright Notice and License

Note: This section is a copy of the W3C Software Copyright Notice and License and could be found at http://www.w3.org/Consortium/Legal/copyright-software-19980720

Copyright © 1994-2001 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved.

http://www.w3.org/Consortium/Legal/

This W3C work (including software, documents, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and modify this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications, that you make:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers. If none exist, then a notice of the following form: "Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/."

1. Content Models and Validation

Editors
Ben Chang, Oracle
Joe Kesselman, IBM
Rezaur Rahman, Intel Corporation

1.1. Overview

This chapter describes the optional DOM Level 3 *Content Model* (*CM*) feature. This module provides a representation for XML content models, e.g., DTDs and XML Schemas, together with operations on the content models, and how such information within the content models could be applied to XML documents used in both the document-editing and CM-editing worlds. It also provides additional tests for well-formedness of XML documents, including Namespace well-formedness. A DOM application can use the `hasFeature` method of the`DOMImplementation` interface to determine whether a given DOM supports these capabilities or not. One feature string for the CM-editing interfaces listed in this section is "CM-EDIT" and another feature string for document-editing interfaces is "CM-DOC".

This chapter interacts strongly with the *Load and Save* chapter, which is also under development in DOM Level 3. Not only will that code serialize/deserialize content models, but it may also wind up defining its well-formedness and validity checks in terms of what is defined in this chapter. In addition, the CM and Load/Save functional areas will share a common error-reporting mechanism allowing user-registered error callbacks. Note that this may not imply that the parser actually calls the DOM's validation code -- it may be able to achieve better performance via its own -- but the appearance to the user should probably be "as if" the DOM has been asked to validate the document, and parsers should probably be able to validate newly loaded documents in terms of a previously loaded DOM CM.

Finally, this chapter will have separate sections to address the needs of the document-editing and CM-editing worlds, along with a section that details overlapping areas such as validation. In this manner, the document-editing world's focuses on editing aspects and usage of information in the CM are made distinct from the CM-editing world's focuses on defining and manipulating the information in the CM.

1.1.1. General Characteristics

In the October 9, 1997 DOM requirements document, the following appeared: "There will be a way to determine the presence of a DTD. There will be a way to add, remove, and change declarations in the underlying DTD (if available). There will be a way to test conformance of all or part of the given document against a DTD (if available)." In later discussions, the following was added, "There will be a way to query element/attribute (and maybe other) declarations in the underlying DTD (if available)," supplementing the primitive support for these in Level 1.

That work was deferred past Level 2, in the hope that XML Schemas would be addressed as well. It is anticipated that lowest common denominator general APIs generated in this chapter can support both DTDs and XML Schemas, and other XML content models down the road.

The kinds of information that a Content Model must make available are mostly self-evident from the definitions of Infoset, DTDs, and XML Schemas. Note that some kinds of information on which the DOM already relies, e.g., default values for attributes, will finally be given a visible representation here, however.

1.1.2. Use Cases and Requirements

The content model referenced in these use cases/requirements is an abstraction and does not refer solely to DTDs or XML Schemas.

For the CM-editing and document-editing worlds, the following use cases and requirements are common to both and could be labeled as the "Validation and Other Common Functionality" section:

Use Cases:

1. CU1. Associating a content model (external and/or internal) with a document, or changing the current association.
2. CU2. Using the same external content model with several documents, without having to reload it.

Requirements:

1. CR1. Validate against the content model.
2. CR2. Retrieve information from content model.
3. CR3. Load an existing content model, perhaps independently from a document.
4. CR4. Being able to determine if a document has a content model associated with it.
5. CR5. Associate a CM with a document and make it the active CM.

Specific to the CM-editing world, the following are use cases and requirements and could be labeled as the "CM-editing" section:

Use Cases:

1. CMU1. Clone/map all or parts of an existing content model to a new or existing content model.
2. CMU2. Save a content model in a separate file. For example, if a DTD can be broken up into reusable pieces, which are then brought in via entity references, these can then be saved in a separate file. Note that the external subset of a DTD, which includes both an internal and external subset, is a special case of dividing a content model into entities.
3. CMU3. Modify an existing content model.
4. CMU4. Create a new content model.
5. CMU5. Partial content model checking. For example, the document need only be validated against a selected portion of the content model.

Requirements:

1. CMR1. View and modify all parts of the content model.
2. CMR2. Validate the content model itself.
3. CMR3. Serialize the content model.

4. CMR4. Clone all or parts of an existing content model.
5. CMR5. Create a new content model object.
6. CMR6. Validate portions of the XML document against the content model.

Specific to the document-editing world, the following are use cases and requirements and could be labeled as the "Document-editing" section:

Use Cases:

1. DU1. For editing documents with an associated content model, provide the guidance necessary so that valid documents can be modified and remain valid.
2. DU2. For editing documents with an associated content model, provide the guidance necessary to transform an invalid document into a valid one.

Requirements:

1. DR1. Be able to determine if the document is well-formed, and if not, be given enough guidance to locate the error.
2. DR2. Be able to determine if the document is namespace well-formed, and if not, be given enough guidance to locate the error.
3. DR3. Be able to determine if the document is valid with respect to its associated content model, and if not, give enough guidance to locate the error.
4. DR4. Be able to determine if specific modifications to a document would make it become invalid.
5. DR5. Retrieve information from all content models. One example might be getting a list of all the defined element names for document editing purposes.

General Issues:

1. I1. Some concerns exist regarding whether a single abstract Content Model structure can successfully represent both namespace-unaware, e.g., DTD, and namespace-aware, e.g., XML Schema, models of document's content. For example, when you ask what elements can be inserted in a specific place, the former will report the element's `QName`, e.g., `foo:bar`, whereas the latter will report its namespace and local name, e.g., `{http://my.namespace}bar`. We have added the `isNamespaceAware` attribute to the generic CM object to help applications determine which of these fields are important, but we are still analyzing this challenge.
2. I2. An XML document may be associated with multiple CMs. We have decided that only one of these is "active" (for validation and guidance) at a time. DOM applications may switch which CM is active, remove CMs that are no longer relevant, or add CMs to the list. If it becomes necessary to simultaneously consult more than one CM, it should be possible to write a "union" CM which provides that capability within this framework.
3. I3. Content model being able to handle more datatypes than strings. Currently, this functionality is not available and should be dealt with in the future.
4. I4. Round-trippability for include/ignore statements and other constructs such as parameter entities, e.g., "macro-like" constructs, will not be supported since no data representation exists to support these constructs without having to re-parse them.
5. I5. Basic interface for a common error handler for both CM and Load/Save. Agreement has been to utilize user-registered callbacks but other details to be worked out.

1.2. Content Model and CM-Editing Interfaces

A list of the proposed Content Model data structures and functions follow, starting off with the data structures and "CM-editing" methods.

Interface *CMModel*

`CMModel` is an abstract object that could map to a DTD, an XML Schema, a database schema, etc. It's a generalized content model object, that has both an internal and external subset. The internal subset would always exist, even if empty, with the external subset (if present) being represented as by an "active" `CMExternalModel` [p.16] . Many `CMExternalModels` could exist, but only one can be specified as "active"; it is also possible that none are "active". The issue of multiple content models is misleading since in this architecture, only one `CMModel` exists, with an internal subset that references the external subset. If the external subset changes to another "acitve" `CMExternalModel`, the internal subset is "fixed up." The CMModel also contains the factory methods needed to create a various types of CMNodes like `CMElementDeclaration` [p.20] , `CMAttributeDeclaration` [p.23] , etc.

IDL Definition

```
interface CMModel : CMNode {
  readonly attribute boolean          isNamespaceAware;
           attribute CMElementDeclaration  rootElementDecl;
  DOMString          getLocation();
  nsElement          getCMNamespace();
  CMNamedNodeMap     getCMNodes();
  boolean            removeNode(in CMNode node);
  boolean            insertBefore(in CMNode newNode,
                                  in CMNode refNode);
  boolean            validate();
  CMElementDeclaration createCMElementDeclaration(inout DOMString namespaceURI,
                                                  in DOMString qualifiedElementName,
                                                  in int contentSpec)
                                        raises(DOMException);
  CMAttributeDeclaration createCMAttributeDeclaration(inout DOMString namespaceURI,
                                                      in DOMString qualifiedName)
                                        raises(DOMException);
  CMNotationDeclaration createCMNotationDeclaration(in DOMString name,
                                                    in DOMString systemIdentifier,
                                                    inout DOMString publicIdentifier)
                                        raises(DOMException);
  CMEntityDeclaration createCMEntityDeclaration(in DOMString name)
                                        raises(DOMException);
  CMChildren         createCMChildren(in unsigned long minOccurs,
                                      in unsigned long maxOccurs,
                                      inout unsigned short operator)
                                        raises(DOMException);
};
```

Attributes

`isNamespaceAware` of type `boolean`, readonly

True if this content model defines the document structure in terms of namespaces and local names; false if the document structure is defined only in terms of `QNames`.

`rootElementDecl` of type `CMElementDeclaration` [p.20]
The root element declaration for the content model. Although a root element is specified in the document instance, when a content model is generated, a user should be able to chose the root element for editing purpose. This is just a placeholder for that element. It could also be null. For validating an XML document, root element must be defined in its active content model. CMModel.rootElementDecl provides access to that root element declaration. This recommendation does not say how to fill in the rootElementdecl. It could be manually done by the user before validating a document, in some cases where possible, the CMModle loader may be able to fill it in etc.

Methods

`createCMAttributeDeclaration`
Creates an attribute declaration. The returned object implements CMAttributeDeclaration interface.

Parameters

`namespaceURI` of type `DOMString`
`qualifiedName` of type `DOMString`
The name of the attribute being declared.

Return Value

`CMAttributeDeclaration` [p.23]	A new CMAttributeDeclaration object with `attributeName` attribute set to input qualifiedname parameter.

Exceptions

`DOMException`	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

`createCMChildren`
Creates a new CMChildren object. The subModels of the CMChildren is build using CMChildren interface methods.

Parameters

`minOccurs` of type `unsigned long`
The minimum occurance for the subModels of this CMChildren.
`maxOccurs` of type `unsigned long`
The maximum occurance for the subModels of this CMChildren.
`operator` of type `unsigned short`
operator of type CHOICE, SEQ or NONE

Return Value

`CMChildren` [p.22]	A new CMChildren object.

Exceptions

`DOMException`	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

`createCMElementDeclaration`

Creates an element declaration for the element type specified. The returned object implements CMElementDeclaration interface.

Parameters

`namespaceURI` of type `DOMString`

`qualifiedElementName` of type `DOMString`

The qualified name of the element type being declared.

`contentSpec` of type `int`

Constant for MIXED, EMPTY, ANY and CHILDREN.

Return Value

`CMElementDeclaration` [p.20]	A new CMElementDeclaration object with `name` attribute set to qualifiedElementName and the `contentType` set to contentSpec. Other attributes of the element declaration are set through CMElementDeclaration interface methods.

Exceptions

`DOMException`	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. DUPLICATE_NAME_ERR:Raised if an element declaration already exists with the same name for a given CMModel.

`createCMEntityDeclaration`

Creates a new entity declaration. The returned object implements CMEntityDeclaration interface.

Parameters

`name` of type `DOMString`

The name of the entity being declared.

Return Value

`CMEntityDeclaration` [p.24]	A new CMNotationDeclaration object with `entityName` attribute set to name.

Exceptions

`DOMException`	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

`createCMNotationDeclaration`

Creates a new notation declaration. The returned object implements CMNotationDeclaration interface.

Parameters

`name` of type `DOMString`

The name of the notation being declared.

`systemIdentifier` of type `DOMString`

The system identifier for the notation declaration.

`publicIdentifier` of type `DOMString`

The public identifier for the notation declaraiton.

Return Value

`CMNotationDeclaration` [p.25]	A new CMNotationDeclaration object with `notationName` attribute set to name.

Exceptions

`DOMException`	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. DUPLICATE_NAME_ERR:Raised if a notation declaration already exists with the same name for a given CMModel.

`getCMNamespace`

Determines namespace of `CMModel`.

Return Value

`nsElement`	Namespace of `CMModel`.

No Parameters

No Exceptions

`getCMNodes`

Returns `CMNode` [p.17] list of all the constituent nodes in the content model.

Return Value

`CMNamedNodeMap` [p.18]	List of all `CMNodes` [p.17] of the content model.

No Parameters

No Exceptions

`getLocation`

Location of the document describing the content model defined in this CMModel.

Return Value

`DOMString`	This method returns a DOMString defining the absolute location from which this document is retrieved including the document name.

No Parameters
No Exceptions

`insertBefore`

Insert `CMNode` [p.17] .

Parameters

`newNode` of type `CMNode` [p.17]
`CMNode` to be inserted.
`refNode` of type `CMNode`
`CMNode` to be inserted before.

Return Value

`boolean`	`Success or failure..`

No Exceptions

`removeNode`

Removes the specified`CMNode` [p.17] .

Parameters

`node` of type `CMNode` [p.17]
`CMNode` to be removed.

Return Value

`boolean`	`Success or failure..`

No Exceptions

`validate`

Determines if a CMModel and CMExternalModel itself is valid, i.e., confirming that it's well-formed and valid per its own formal grammar. Note that within a CMModel, a pointer to a CMExternalModel can exist.

Return Value

`boolean`	Is the CM valid?

No Parameters
No Exceptions

Interface *CMExternalModel*

`CMExternalModel` is an abstract object that could map to a DTD, an XML Schema, a database schema, etc. It's a generalized content model object that is not bound to a particular XML document.

IDL Definition

```
interface CMExternalModel : CMModel {
};
```

Interface *CMNode*

`CMNode`is analogous to a `Node` in the Core DOM, e.g., an element declaration. This can exist for both `CMExternalModel` [p.16] and `CMModel` [p.12] . It should be able to handle constructs such as comments and processing instructions.

Opaque.

IDL Definition

```
interface CMNode {
  const unsigned short      CM_ELEMENT_DECLARATION         = 1;
  const unsigned short      CM_ATTRIBUTE_DECLARATION       = 2;
  const unsigned short      CM_NOTATION_DECLARATION        = 3;
  const unsigned short      CM_ENTITY_DECLARATION          = 4;
  const unsigned short      CM_CHILDREN                    = 5;
  const unsigned short      CM_MODEL                       = 6;
  const unsigned short      CM_EXTERNALMODEL               = 7;
  readonly attribute unsigned short   cmNodeType;
           attribute CMModel          ownerCMModel;
           attribute DOMString        nodeName;
           attribute DOMString        prefix;
           attribute DOMString        localName;
           attribute DOMString        namespaceURI;
  CMNode             clone();
};
```

Constant *CM_ELEMENT_DECLARATION*
The node is an `CMElementDeclaration` [p.20] .
Constant *CM_ATTRIBUTE_DECLARATION*
The node is an `CMAttributeDeclaration` [p.23] .
Constant *CM_NOTATION_DECLARATION*
The node is a `CMNotationDeclaration` [p.25] .
Constant *CM_ENTITY_DECLARATION*
The node is an `CMEntityDeclaration` [p.24] .
Constant *CM_CHILDREN*
The node is a `CMChildren` [p.22] .
Constant *CM_MODEL*
The node is a `CMModel` [p.12] .
Constant *CM_EXTERNALMODEL*
The node is a `CMExternalModel` [p.16] .
Attributes
`cmNodeType` of type `unsigned short`, readonly
A code representing the underlying object as defined above.
`localName` of type `DOMString`
Returns the local part of the `qualified name` of this CMNode.

`namespaceURI` of type `DOMString`
The namespace URI of this node, or null if it is unspecified.
`nodeName` of type `DOMString`
The `qualified name` of this CMNode depending on the CMNode type.
`ownerCMModel` of type `CMModel` [p.12]
The `CMModel` [p.12] object associated with this CMNode. For a node of type `CM_MODEL`, this is `null`.
`prefix` of type `DOMString`
The namespace prefix of this node, or null if it is unspecified.

Methods
`clone`
Creates a copy of `CMNode`.
Return Value

`CMNode` [p.17]	Cloned `CMNode`.

No Parameters
No Exceptions

Interface *CMNodeList*

`CMNodeList` is the CM analogue to `NodeList`; the document order is meaningful, as opposed to `CMNamedNodeMap` [p.18] .

IDL Definition

```
interface CMNodeList {
};
```

Interface *CMNamedNodeMap*

`CMNamedNodeMap` is the CM analogue to `NamedNodeMap`. The order is not meaningful.

IDL Definition

```
interface CMNamedNodeMap {
};
```

Interface *CMDataType*

The primitive datatypes supported by base DOM CM implementation is: `string` type only.

IDL Definition

```
interface CMDataType {
  const short               STRING_DATATYPE                = 1;
  short              getCMPrimitiveType();
};
```

Constant *STRING_DATATYPE*

code representing the `string` data type as defined in XML Schema Datatypes.

Methods

`getCMPrimitiveType`

Returns one of the enumerated code representing the primitive data type.

Return Value

`short`	code representing the primitive type of the attached data item.

No Parameters

No Exceptions

Interface *CMPrimitiveType*

The primitive types supported by optional DOM CM implelementations. A DOM application can use the hasFeature method of the DOMImplementation interface to determine whether this interface is supported or not. The feature string for all the interfaces listed in this section is "CMPTYPES" and the version is "3.0".

IDL Definition

```
interface CMPrimitiveType : CMDataType {
  const short              BOOLEAN_DATATYPE              = 2;
  const short              FLOAT_DATATYPE                = 3;
  const short              DOUBLE_DATATYPE               = 4;
  const short              DECIMAL_DATATYPE              = 5;
  const short              HEXBINARY_DATATYPE            = 6;
  const short              BASE64BINARY_DATATYPE         = 7;
  const short              ANYURI_DATATYPE               = 8;
  const short              QNAME_DATATYPE                = 9;
  const short              DURATION_DATATYPE             = 10;
  const short              DATETIME_DATATYPE             = 11;
  const short              DATE_DATATYPE                 = 12;
  const short              TIME_DATATYPE                 = 13;
  const short              YEARMONTH_DATATYPE            = 14;
  const short              YEAR_DATATYPE                 = 15;
  const short              MONTHDAY_DATATYPE             = 16;
  const short              DAY_DATATYPE                  = 17;
  const short              MONTH_DATATYPE                = 18;
  const short              NOTATION_DATATYPE             = 19;
           attribute decimal          lowValue;
           attribute decimal          highValue;
};
```

Constant *BOOLEAN_DATATYPE*

code representing the `boolean` data type as defined in XML Schema Datatypes.

Constant *FLOAT_DATATYPE*

code representing the `float` data type as defined in XML Schema Datatypes.

Constant *DOUBLE_DATATYPE*

code representing the `double` data type as defined in XML Schema Datatypes.

Constant *DECIMAL_DATATYPE*
code representing a `decimal` data type as defined in XML Schema Datatypes.
Constant *HEXBINARY_DATATYPE*
code representing a `hexbinary` data type as defined in XML Schema Datatypes.
Constant *BASE64BINARY_DATATYPE*
code representing a `base64binary` data type as defined in XML Schema Datatypes.
Constant *ANYURI_DATATYPE*
code representing an `uri reference` data type as defined in XML Schema Datatypes.

Note: @@uriReference is no longer part of the XML Schema PR draft.

Constant *QNAME_DATATYPE*
code representing an `XML qualified name` data type as defined in XML Schema Datatypes.
Constant *DURATION_DATATYPE*
code representing a `duration` data type as defined in XML Schema Datatypes.
Constant *DATETIME_DATATYPE*
code representing a`datetime` data type as defined in XML Schema Datatypes.
Constant *DATE_DATATYPE*
code representing a`date` data type as defined in XML Schema Datatypes.
Constant *TIME_DATATYPE*
code representing a `time` data type as defined in XML Schema Datatypes.
Constant *YEARMONTH_DATATYPE*
code representing a `yearmonth` data type as defined in XML Schema Datatypes.
Constant *YEAR_DATATYPE*
code representing a `year` data type as defined in XML Schema Datatypes.
Constant *MONTHDAY_DATATYPE*
code representing a `monthday` data type as defined in XML Schema Datatypes.
Constant *DAY_DATATYPE*
code representing a `day` data type as defined in XML Schema Datatypes.
Constant *MONTH_DATATYPE*
code representing a `month` data type as defined in XML Schema Datatypes.
Constant *NOTATION_DATATYPE*
code representing a`NOTATION`data type as defined in XML Schema Datatypes.

Attributes

`highValue` of type `decimal`
The high value for a primitive DECIMAL_DATATYPE in the value range.
`lowValue` of type `decimal`
The low value for a primitive DECIMAL_DATATYPE in the value range.

Interface *CMElementDeclaration*

The element name along with the content specification in the context of a `CMNode` [p.17] .

IDL Definition

```
interface CMElementDeclaration : CMNode {
           attribute CMDataType       elementType;
  readonly attribute boolean          isPCDataOnly;
           attribute DOMString        tagName;
  int                getContentType();
  CMChildren         getCMChildren();
  CMNamedNodeMap     getCMAttributes();
  CMNamedNodeMap     getCMGrandChildren();
};
```

Attributes

`elementType` of type `CMDataType` [p.18]
: Datatype of the element.

`isPCDataOnly` of type `boolean`, readonly
: Boolean defining whether the element type contains child elements and PCDATA or PCDATA only for mixed element types. True if the element is of type PCDATA only. Relevant only for mixed content type elements.

`tagName` of type `DOMString`
: tagName of the element being declared.

Methods

`getCMAttributes`
: Returns a `CMNamedNodeMap` [p.18] containing `CMAttributeDeclarations` [p.23] for all the attributes that can appear on this type of element.

Return Value

`CMNamedNodeMap` [p.18]	Attributes list for this `CMNode` [p.17] .

No Parameters

No Exceptions

`getCMChildren`
: Gets content model of element.

Return Value

`CMChildren` [p.22]	Content model of element.

No Parameters

No Exceptions

`getCMGrandChildren`
: Returns a `CMNamedNodeMap` [p.18] containing `CMElementDeclarations` for all the `Elements` that can appear as children of this type of element. Note that which ones can actually appear, and in what order, is defined by the `CMChildren` [p.22] .

Return Value

`CMNamedNodeMap` [p.18]	Children list for this `CMNode` [p.17] .

No Parameters
No Exceptions

`getContentType`

Gets content type, e.g., empty, any, mixed, elements, PCDATA, of an element.

Return Value

`int`	Content type constant.

No Parameters
No Exceptions

Interface *CMChildren*

The content model of a declared element.

IDL Definition

```
interface CMChildren : CMNode {
  const unsigned long       UNBOUNDED                      = MAX_LONG;
  const unsigned short      NONE                           = 0;
  const unsigned short      SEQUENCE                       = 1;
  const unsigned short      CHOICE                         = 2;
           attribute unsigned short   listOperator;
           attribute unsigned long    minOccurs;
           attribute unsigned long    maxOccurs;
           attribute CMNodeList       subModels;
  CMNode             removeCMNode(in unsigned long nodeIndex);
  int                insertCMNode(in unsigned long nodeIndex,
                                  in CMNode newNode);
  int                appendCMNode(in CMNode newNode);
};
```

Constant *UNBOUNDED*

Signifies unbounded upper limit. The MAX_LONG value is the maximum value of an unsigned long integer for a given language binding.

Constant *NONE*

No operators defined on the `subModels`. This is usually the case where the subModels contain a single element declaration.

Constant *SEQUENCE*

This constant value signifies a sequence operator ",".

Constant *CHOICE*

This constant value signifies a choice operator "|".

Attributes

`listOperator` of type `unsigned short`

One of CHOICE or SEQUENCE. The operator is applied to all the components(CMNodes) in the the `subModels`. For example, if the list operator is CHOICE and the components in subModels are a, b and c then the content model for the element being declared is (a|b|c)

`maxOccurs` of type `unsigned long`

maximum occurrence for this content particle. Valid values are from 0 to UNBOUNDED.

`minOccurs` of type `unsigned long`
min occurrence for this content particle. Valid values are from 0 to UNBOUNDED.
`subModels` of type `CMNodeList` [p.18]
Additional `CMNode` [p.17] s in which the element can be defined.

Methods

`appendCMNode`
Appends a new node to the end of the list representing the`subModels`.
Parameters
`newNode` of type `CMNode` [p.17]
The new node to be appended.
Return Value

`int`	the length of the `subModels`.

No Exceptions

`insertCMNode`
Inserts a new node at a position in the submodel referred to by the nodeIndex. Node already exisiting in the list is moved as needed.
Parameters
`nodeIndex` of type `unsigned long`
The position of where the newNode is inserted.
`newNode` of type `CMNode` [p.17]
The new node to be inserted.
Return Value

`int`	The index value at which it is inserted. If the nodeIndex is outside the bound of the `subModels` list, the item is inserted at the back of the list.

No Exceptions

`removeCMNode`
Removes the CMNode at the indicated index position in the submodel.
Parameters
`nodeIndex` of type `unsigned long`
Index of the node being removed.
Return Value

`CMNode` [p.17]	The node removed is returned as a result of this method call. The method returns `null` if the index is outside the bounds of the `subModels` list.

No Exceptions

Interface *CMAttributeDeclaration*

An attribute declaration in the context of a `CMNode` [p.17] .

IDL Definition

```
interface CMAttributeDeclaration : CMNode {
  const short               NO_VALUE_CONSTRAINT            = 0;
  const short               DEFAULT_VALUE_CONSTRAINT       = 1;
  const short               FIXED_VALUE_CONSTRAINT         = 2;
           attribute DOMString        attrName;
           attribute CMDataType       attrType;
           attribute DOMString        attributeValue;
           attribute DOMString        enumAttr;
           attribute CMNodeList       ownerElement;
           attribute short            constraintType;
};
```

Constant *NO_VALUE_CONSTRAINT*
Describes that the attribute does not have any value constraint.
Constant *DEFAULT_VALUE_CONSTRAINT*
Indicates that the there is a default value constraint.
Constant *FIXED_VALUE_CONSTRAINT*
Indicates that there is a fixed value constraint for this attribute.
Attributes
`attrName` of type `DOMString`
Name of the attribute.
`attrType` of type `CMDataType` [p.18]
Datatype of the attribute.
`attributeValue` of type `DOMString`
Default value.
`constraintType` of type `short`
Constraint type if any for this attribute.
`enumAttr` of type `DOMString`
Enumeration of attribute.
`ownerElement` of type `CMNodeList` [p.18]
Owner element CMNode of attribute.

Interface *CMEntityDeclaration*

Models a general entity declaration in a content model.

(***ED:*** The content model does not handle any parameter entity. It is assumed that the parameter entiites are expanded by the implementation as the content model is built.)
IDL Definition

```
interface CMEntityDeclaration : CMNode {
  const short               INTERNAL_ENTITY                = 1;
  const short               EXTERNAL_ENTITY                = 2;
           attribute short            entityType;
           attribute DOMString        entityName;
           attribute DOMString        entityValue;
           attribute DOMString        systemId;
           attribute DOMString        publicId;
           attribute DOMString        notationName;
};
```

Constant *INTERNAL_ENTITY*
constant defining an internal entity.

Constant *EXTERNAL_ENTITY*
constant defining an external entity.

Attributes

`entityName` of type `DOMString`
The name of the declared general entity.

`entityType` of type `short`
One of the INTERNAL_ENTITY or EXTERNAL_ENTITY.

`entityValue` of type `DOMString`
The replacement text for the internal entity. The entity references within the replacement text is kept intact. For entity of type EXTERNAL_ENTITY this is `null`.

`notationName` of type `DOMString`
For unparsed entities, the name of the notation declaration for the entity. For parsed entities, this is `null`.

`publicId` of type `DOMString`
The public identifier associated with the entity, if specified. If the public identifier was not specified, this is `null`.

`systemId` of type `DOMString`
The system identifier associated with the entity, if specified. If the system identifier was not specified, this is `null`.

Interface *CMNotationDeclaration*

This interface represents a notation declaration.

IDL Definition

```
interface CMNotationDeclaration : CMNode {
           attribute DOMString        notationName;
           attribute DOMString        systemId;
           attribute DOMString        publicId;
};
```

Attributes

`notationName` of type `DOMString`
The name of this notation declaration.

`publicId` of type `DOMString`
The string representing the public identifier for this notation declaration.

`systemId` of type `DOMString`
the URI representing the system identifier for the notation declaration, if present, null otherwise.

1.3. Validation and Other Interfaces

This section contains "Validation and Other" methods common to both the document-editing and CM-editing worlds (includes `Document` [p.26] , `DOMImplementation`, and `DOMErrorHandler` [p.40] methods).

Interface *Document*

The `setErrorHandler` method is off of the `Document` interface.

IDL Definition

```
interface Document {
  void               setErrorHandler(in DOMErrorHandler handler);
};
```

Methods

`setErrorHandler`
Allow an application to register an error event handler.
Parameters
`handler` of type `DOMErrorHandler` [p.40]
The error handler
No Return Value
No Exceptions

Interface *DocumentCM*

This interface extends the `Document` [p.26] interface with additional methods for both document and CM editing.

IDL Definition

```
interface DocumentCM : Document {
  const short               WF_CHECK                      = 1;
  const short               NS_WF_CHECK                   = 2;
  const short               PARTIAL_VALIDITY_CHECK        = 3;
  const short               STRICT_VALIDITY_CHECK         = 4;
           attribute boolean          continuousValidityChecking;
           attribute short            wfValidityCheckLevel;
  int                numCMs();
  CMModel            getInternalCM();
  CMNodeList         getCMs();
  CMModel            getActiveCM();
  void               addCM(in CMModel cm);
  void               removeCM(in CMModel cm);
  boolean            activateCM(in CMModel cm);
};
```

Constant *WF_CHECK*
Check for well-formedness of the document.
Constant *NS_WF_CHECK*
Check for namespace well-formedness includes WF_CHECK.
Constant *PARTIAL_VALIDITY_CHECK*
Checks for whether the document is partially valid. It includes NS_WF_CHECK.
A document is said to be *partially valid* if it contains elments/attributes for which an element/attribute declaration has not been made in the active content model. However, if the element or the attribute has a declaration in the content model, it must be valid with respect to those declarations.
Constant *STRICT_VALIDITY_CHECK*
Checks for strict validity of the document with respect to active CM which by defiition includes NS_WF_CHECK.
Attributes
`continuousValidityChecking` of type `boolean`
An attribute specifying whether continuous checking for the validity of the document is enforced or not. When set to true the implementation is free to raise the VALIDATION_ERR exception on DOM operations that would make the document invalid with respect to "partial validity". This attribute is false by default.
(***ED:*** Add VALIDATION_ERR code to the list of constants in DOMException.)
`wfValidityCheckLevel` of type `short`
This attribute defines the level at which the validity and welformedness testing is done by the `isValid` method.
Methods
`activateCM`
Make the given `CMModel` [p.12] active. Note that if a user wants to activate one CM to get default attribute values and then activate another to do validation, a user can do that; however, only one CM is active at a time. In case where an attribute is declared in an internal subset and corresponding `ownerElement` points to `CMElementDeclaration` [p.20] defined in an external subset, changing active CM will cause the `ownerElement` to be re-computed. If the owner element is not defined in the newly active CM, the `ownerElement` will be an empty node list.
Parameters
`cm` of type `CMModel` [p.12]
CM to be active for the document. The `CMModel` points to a list of `CMExternalModel` [p.16] s; with this call, only the specified CM will be active.
Return Value

`boolean`	True if the `CMModel` has already been associated with the document using `addCM()`; false if not.

No Exceptions
`addCM`
Associate a `CMModel` [p.12] with a document. Can be invoked multiple times to result in a list of `CMExternalModel` [p.16] s. Note that only one sole internal `CMModel` is associated with the document, however, and that only one of the possible list of

CMExternalModels is active at any one time.
Parameters
cm of type CMModel [p.12]
CM to be associated with the document.
No Return Value
No Exceptions

getActiveCM
Find the active CMExternalModel [p.16] for a document.
Return Value

CMModel [p.12]	CMModel with a pointer to the active CMExternalModel [p.16] of document.

No Parameters
No Exceptions

getCMs
Obtains list of CMNodes [p.17] of typeCM_EXTERNALMODELs associated with the document.This list arises when addCM() is invoked.
Return Value

CMNodeList [p.18]	A list of CMExternalModel [p.16] s associated with a document.

No Parameters
No Exceptions

getInternalCM
Find the sole CMModel [p.12] of a document. Only one CMModel may be associated with the document.
Return Value

CMModel [p.12]	CMModel.

No Parameters
No Exceptions

numCMs
Determines number of CMExternalModel [p.16] s associated with the document. Only one CMModel [p.12] can be associated with the document, but it may point to a list of CMExternalModels.
Return Value

int	Non-negative number of external CM objects.

No Parameters
No Exceptions

`removeCM`
Removes a CM associated with a document; actually removes a `CMExternalModel` [p.16] . Can be invoked multiple times to remove a number of these in the list of `CMExternalModels`.
Parameters
`cm` of type `CMModel` [p.12]
CM to be removed.
No Return Value
No Exceptions

Interface *DOMImplementationCM*

This interface extends the `DOMImplementation` interface with additional methods.

IDL Definition

```
interface DOMImplementationCM : DOMImplementation {
  CMModel            createCM();
  CMExternalModel    createExternalCM();
};
```

Methods
`createCM`
Creates a CMModel.
Return Value

`CMModel` [p.12]	A NULL return indicates failure.

No Parameters
No Exceptions

`createExternalCM`
Creates a CMExternalModel.
Return Value

`CMExternalModel` [p.16]	A NULL return indicates failure.

No Parameters
No Exceptions

1.4. Document-Editing Interfaces

This section contains "Document-editing" methods (includes `Node`, `Element`, `Text` and `Document` [p.26] methods).

Interface *NodeCM*

This interface extends the `Node` interface with additional methods for guided document editing.

IDL Definition

```
interface NodeCM : Node {
  boolean            canInsertBefore(in Node newChild,
                                     in Node refChild)
                                        raises(DOMException);
  boolean            canRemoveChild(in Node oldChild)
                                        raises(DOMException);
  boolean            canReplaceChild(in Node newChild,
                                     in Node oldChild)
                                        raises(DOMException);
  boolean            canAppendChild(in Node newChild)
                                        raises(DOMException);
  boolean            isValid()
                                        raises(DOMException);
};
```

Methods

`canAppendChild`

Has the same args as `AppendChild`.

Parameters

`newChild` of type `Node`

`Node` to be appended.

Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canInsertBefore`

Determines whether the `Node::InsertBefore` operation would make this document invalid with respect to the currently active CM. ISSUE: Describe "valid" when referring to partially completed documents.

Parameters

`newChild` of type `Node`

`Node` to be inserted.

`refChild` of type `Node`

Reference `Node`.

Return Value

`boolean`	A boolean that is true if the `Node::InsertBefore` operation is allowed.

Exceptions

`DOMException`	DOMException.

`canRemoveChild`

Has the same args as `RemoveChild`.

Parameters

`oldChild` of type `Node`

`Node` to be removed.

Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canReplaceChild`

Has the same args as `ReplaceChild`.

Parameters

`newChild` of type `Node`

New `Node`.

`oldChild` of type `Node`

`Node` to be replaced.

Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`isValid`

Determines if the Node is valid relative to currently active CM.

Return Value

`boolean`	True if the node is valid/well-formed in the current context and check level defined by wfValidityCheckLevel, false if not.

Exceptions

`DOMException`	NO_CM_AVAILABLE: Exception is raised if the DocumentCM related to this node does not have any activeCM and wfValidityCheckLevel is set to STRICT_VALIDITY_CHECK.

No Parameters

Interface *ElementCM*

This interface extends the `Element` interface with additional methods for guided document editing.

IDL Definition

```
interface ElementCM : Element,NodeCM {
  int                contentType();
  CMElementDeclaration getElementDeclaration()
                                        raises(DOMException);
  boolean            canSetAttribute(in DOMString attrname,
                                     in DOMString attrval);
  boolean            canSetAttributeNode(in Node node);
  boolean            canSetAttributeNodeNS(in Node node);
  boolean            canSetAttributeNS(in DOMString attrname,
                                       in DOMString attrval,
                                       in DOMString namespaceURI,
                                       in DOMString localName);
  boolean            canRemoveAttribute(in DOMString attrname);
  boolean            canRemoveAttributeNS(in DOMString attrname,
                                          inout DOMString namespaceURI);
  boolean            canRemoveAttributeNode(in Node node);
};
```

Methods

`canRemoveAttribute`

Verifies if an attribute by the given name can be removed.

Parameters

`attrname` of type `DOMString`

Name of attribute.

Return Value

`boolean`	true or false.

No Exceptions

`canRemoveAttributeNS`

Verifies if an attribute by the given name and namespace can be removed.

Parameters

`attrname` of type `DOMString`

Qualified name of the attribute to be removed.

`namespaceURI` of type `DOMString`

The namespace URI of the attribute to remove.

Return Value

`boolean`	true or false.

No Exceptions

`canRemoveAttributeNode`

Determines if an attribute node can be removed.

Parameters

`node` of type `Node`

The `Attr` node to remove from the attribute list.

Return Value

`boolean`	true or false.

No Exceptions

`canSetAttribute`

Determines if the value for specified attribute can be set.

Parameters

`attrname` of type `DOMString`

Name of attribute.

`attrval` of type `DOMString`

Value to be assigned to the attribute.

Return Value

`boolean`	true or false.

No Exceptions

`canSetAttributeNS`

Determines if the attribute with given namespace and local name can be created if not already present in the attribute list of the element. If the attribute with same local name and namespaceURI is already present in the elements attribute list it sets the value of the attribute and its prefix to the new value. See DOM core `setAttributeNS`.

Parameters

`attrname` of type `DOMString`

Name of attribute.

`attrval` of type `DOMString`

Value to be assigned to the attribute.

`namespaceURI` of type `DOMString`

`namespaceURI` of namespace.

`localName` of type `DOMString`

`localName` of namespace.

Return Value

`boolean`	Success or failure.

No Exceptions

`canSetAttributeNode`

Determines if attribute node can be added.

Parameters

`node` of type `Node`

`Node` in which the attribute can possibly be set.

Return Value

`boolean`	Success or failure.

No Exceptions

`canSetAttributeNodeNS`

Determines if the attribute node with the given namespace can be added.

Parameters

`node` of type `Node`

The `Attr` to be added to the attribute list.

Return Value

`boolean`	Success or failure.

No Exceptions

`contentType`

Determines element content type.

Return Value

`int`	Constant for mixed, empty, any, etc.

No Parameters

No Exceptions

`getElementDeclaration`

gets the CM editing object describing this element

Return Value

`CMElementDeclaration` [p.20]	CMElementDeclaration object

Exceptions

`DOMException`	If no DTD is present raises this exception

No Parameters

Interface *CharacterDataCM*

This interface extends the `CharacterData` interface with additional methods for document editing.

IDL Definition

```
interface CharacterDataCM : Text,NodeCM {
  boolean            isWhitespaceOnly();
  boolean            canSetData(in unsigned long offset,
                                in DOMString arg)
                                       raises(DOMException);
  boolean            canAppendData(in DOMString arg)
                                       raises(DOMException);
  boolean            canReplaceData(in unsigned long offset,
                                    in unsigned long count,
                                    in DOMString arg)
                                       raises(DOMException);
  boolean            canInsertData(in unsigned long offset,
                                   in DOMString arg)
                                       raises(DOMException);
  boolean            canDeleteData(in unsigned long offset,
                                   in DOMString arg)
                                       raises(DOMException);
};
```

Methods

`canAppendData`

Determines if data can be appended.

Parameters

`arg` of type `DOMString`

Argument to be appended.

Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canDeleteData`

Determines if data can be deleted.

Parameters

`offset` of type `unsigned long`

Offset.

`arg` of type `DOMString`

Argument to be set.

Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canInsertData`
Determines if data can be inserted.
Parameters
`offset` of type `unsigned long`
Offset.
`arg` of type `DOMString`
Argument to be set.
Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canReplaceData`
Determines if data can be replaced.
Parameters
`offset` of type `unsigned long`
Offset.
`count` of type `unsigned long`
Replacement.
`arg` of type `DOMString`
Argument to be set.
Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`canSetData`
Determines if data can be set.
Parameters
`offset` of type `unsigned long`
Offset.
`arg` of type `DOMString`
Argument to be set.
Return Value

`boolean`	Success or failure.

Exceptions

`DOMException`	DOMException.

`isWhitespaceOnly`
Determines if content is only whitespace.
Return Value

`boolean`	True if content only whitespace; false for non-whitespace if it is a text node in element content.

No Parameters
No Exceptions

Interface *DocumentTypeCM*

This interface extends the `DocumentType` interface with additional methods for document editing.

IDL Definition

```
interface DocumentTypeCM : DocumentType,NodeCM {
  boolean            isElementDefined(in DOMString elemTypeName);
  boolean            isElementDefinedNS(in DOMString elemTypeName,
                                        in DOMString namespaceURI,
                                        in DOMString localName);
  boolean            isAttributeDefined(in DOMString elemTypeName,
                                        in DOMString attrName);
  boolean            isAttributeDefinedNS(in DOMString elemTypeName,
                                          in DOMString attrName,
                                          in DOMString namespaceURI,
                                          in DOMString localName);
  boolean            isEntityDefined(in DOMString entName);
};
```

Methods

`isAttributeDefined`
Determines if this attribute is defined for this element in the currently active CM.
Parameters
`elemTypeName` of type `DOMString`
Name of the element.
`attrName` of type `DOMString`
Name of the attribute.
Return Value

`boolean`	Success or failure.

No Exceptions

`isAttributeDefinedNS`

Determines if this attribute's namespace is defined in the currently active CM.

Parameters

`elemTypeName` of type `DOMString`
Name of element.

`attrName` of type `DOMString`
Name of attribute.

`namespaceURI` of type `DOMString`
`namespaceURI` of namespace.

`localName` of type `DOMString`
`localName` of namespace.

Return Value

`boolean`	Success or failure.

No Exceptions

`isElementDefined`

Determines if this element is defined in the currently active CM.

Parameters

`elemTypeName` of type `DOMString`
Name of element.

Return Value

`boolean`	Success or failure.

No Exceptions

`isElementDefinedNS`

Determines if this element's namespace is defined in the currently active CM.

Parameters

`elemTypeName` of type `DOMString`
Name of element.

`namespaceURI` of type `DOMString`
`namespaceURI` of namespace.

`localName` of type `DOMString`
`localName` of namespace.

Return Value

`boolean`	Success or failure.

No Exceptions

`isEntityDefined`

Determines if an entity is defined in the document.

ISSUE: Should methods be added to the DocumentTypeCM for the complete list of defined elements and for a particular element type, the complete list of defined attributes.

These two methods might return a list of strings which is a type not yet described in the DOM spec.

Parameters

`entName` of type `DOMString`

Name of entity.

Return Value

`boolean`	Success or failure.

No Exceptions

Interface *AttributeCM*

This interface extends `Attr` to provide guided editing of an XML document.

IDL Definition

```
interface AttributeCM : Attr,NodeCM {
  CMAttributeDeclaration getAttributeDeclaration();
  CMNotationDeclaration getNotation()
                                        raises(DOMException);
};
```

Methods

`getAttributeDeclaration`

returns the corresponding attribute declaration in the content model.

Return Value

`CMAttributeDeclaration` [p.23]	The attribute declaration corresponding to this attribute

No Parameters

No Exceptions

`getNotation`

Returns the notation declaration for the attributes defined of type NOTATION.

Return Value

`CMNotationDeclaration` [p.25]	Returns the notation declaration for this attribute if the type is of notation type, null otherwise.

Exceptions

`DOMException`	DOMException

No Parameters

1.5. DOM Error Handler Interfaces

This section contains DOM error handling interfaces.

Interface *DOMErrorHandler*

Basic interface for DOM error handlers. If an application needs to implement customized error handling for DOM such as CM or Load/Save, it must implement this interface and then register an instance using the `setErrorHandler` method. All errors and warnings will then be reported through this interface. Application writers can override the methods in a subclass to take user-specified actions.

IDL Definition

```
interface DOMErrorHandler {
  void               warning(in DOMLocator where,
                             in DOMString how,
                             in DOMString why)
                                        raises(DOMSystemException);
  void               fatalError(in DOMLocator where,
                                in DOMString how,
                                in DOMString why)
                                        raises(DOMSystemException);
  void               error(in DOMLocator where,
                           in DOMString how,
                           in DOMString why)
                                        raises(DOMSystemException);
};
```

Methods

`error`

Receive notification of a recoverable error per section 1.2 of the W3C XML 1.0 recommendation. The default behavior if the user doesn't register a handler is to report conditions that are not fatal errors, and allow the calling application to continue processing.

Parameters

`where` of type `DOMLocator` [p.41]

Location of the error, which could be either a source position in the case of loading, or a node reference for later validation. The public ID and system ID for the error location could be some of the information.

`how` of type `DOMString`

How the error occurred.

`why` of type `DOMString`

Why the error occurred.

Exceptions

`DOMSystemException`	A subclass of DOMException.

No Return Value

`fatalError`
Report a fatal, non-recoverable CM or Load/Save error per section 1.2 of the W3C XML 1.0 recommendation. The default behavior if the user doesn't register a handler is to throw a DOMSystemException and stop all further processing.
Parameters
`where` of type `DOMLocator` [p.41]
Location of the fatal error, which could be either a source position in the case of loading, or a node reference for later validation. The public ID and system ID for the error location could be some of the information.
`how` of type `DOMString`
How the fatal error occurred.
`why` of type `DOMString`
Why the fatal error occurred.
Exceptions

`DOMSystemException`	A subclass of DOMException.

No Return Value

`warning`
Receive notification of a warning per the W3C XML 1.0 recommendation. The default behavior if the user doesn't register a handler is to report conditions that are not errors or fatal errors, and then allow the calling application to continue even after invoking this method.
Parameters
`where` of type `DOMLocator` [p.41]
Location of the warning, which could be either a source position in the case of loading, or a node reference for later validation. The public ID and system ID for the error location could be some of the information.
`how` of type `DOMString`
How the warning occurred.
`why` of type `DOMString`
Why the warning occurred.
Exceptions

`DOMSystemException`	A subclass of DOMException.

No Return Value

Interface *DOMLocator*

This interface provides document location information and is similar to a SAX locator object.

IDL Definition

```
interface DOMLocator {
  int                getColumnNumber();
  int                getLineNumber();
  DOMString          getPublicID();
  DOMString          getSystemID();
  Node               getNode();
};
```

Methods

`getColumnNumber`

Return the column number.

Return Value

`int`	The column number, or -1 if none is available.

No Parameters

No Exceptions

`getLineNumber`

Return the line number.

Return Value

`int`	The line number, or -1 if none is available.

No Parameters

No Exceptions

`getNode`

Return the Node.

Return Value

`Node`	The NODE, or null if none is available.

No Parameters

No Exceptions

`getPublicID`

Return the public identifier.

Return Value

`DOMString`	A string containing the public identifier, or null if none is available.

No Parameters

No Exceptions

`getSystemID`

Return the system identifier.

Return Value

`DOMString` A string containing the system identifier, or null if none is available.

No Parameters
No Exceptions

1.6. Editing and Generating a Content Model

Editing and generating a content model falls in the CM-editing world. The most obvious requirement for this set of requirements is for tools that author content models, either under user control, i.e., explicitly designed document types, or generated from other representations. The latter class includes transcoding tools, e.g., synthesizing an XML representation to match a database schema.

It's important to note here that a DTD's "internal subset" is part of the Content Model, yet is loaded, stored, and maintained as part of the individual document instance. This implies that even tools which do not want to let users change the definition of the Document Type may need to support editing operations upon this portion of the CM. It also means that our representation of the CM must be aware of where each portion of its content resides, so that when the serializer processes this document it can write out just the internal subset. A similar issue may arise with external parsed entities, or if schemas introduce the ability to reference other schemas. Finally, the internal-subset case suggests that we may want at least a two-level representation of content models, so a single DOM representation of a DTD can be shared among several documents, each potentially also having its own internal subset; it's possible that entity layering may be represented the same way.

The API for altering the content model may also be the CM's official interface with parsers. One of the ongoing problems in the DOM is that there is some information which must currently be created via completely undocumented mechanisms, which limits the ability to mix and match DOMs and parsers. Given that specialized DOMs are going to become more common (sub-classed, or wrappers around other kinds of storage, or optimized for specific tasks), we must avoid that situation and provide a "builder" API. Particular pairs of DOMs and parsers may bypass it, but it's required as a portability mechanism.

Note that several of these applications require that a CM be able to be created, loaded, and manipulated without/before being bound to a specific Document. A related issue is that we'd want to be able to share a single representation of a CM among several documents, both for storage efficiency and so that changes in the CM can quickly be tested by validating it against a set of known-good documents. Similarly, there is a known problem in DOM Level 2 where we assume that the DocumentType will be created before the Document, which is fine for newly-constructed documents but not a good match for the order in which an XML parser encounters this data; being able to "rebind" a Document to a new CM, after it has been created may be desirable.

As noted earlier, questions about whether one can alter the content of the CM via its syntax, via higher-level abstractions, or both, exist. It's also worth noting that many of the editing concepts from the Document tree still apply; users should probably be able to clone part of a CM, remove and re-insert parts, and so on.

1.7. Content Model-directed Document Manipulation

In addition to using the content model to validate a document instance, applications would like to be able to use it to guide construction and editing of documents, which falls into the document-editing world. Examples of this sort of guided editing already exist, and are becoming more common. The necessary queries can be phrased in several ways, the most useful of which may be a combination of "what does the DTD allow me to insert here" and "if I insert this here, will the document still be valid". The former is better suited to presentation to humans via a user interface, and when taken together with sub-tree validation may subsume the latter.

It has been proposed that in addition to asking questions about specific parts of the content model, there should be a reasonable way to obtain a list of all the defined symbols of a given type (element, attribute, entity) independent of whether they're valid in a given location; that might be useful in building a list in a user-interface, which could then be updated to reflect which of these are relevant for the program's current state.

Remember that namespaces also weigh in on this issue, in the case of attributes, a "can-this-go-there" may prompt a namespace-well-formedness check and warn you if you're about to conflict with or overwrite another attribute with the same namespaceURI/localName but different prefix... or same nodeName but different namespaceURI.

As mentioned above, we have to deal with the fact that the shortest distance between two valid documents may be through an invalid one. Users may want to know several levels of detail (all the possible children, those which would be valid given what precedes this point, those which would be valid given both preceding and following siblings). Also, once XML Schemas introduce context sensitive validity, we may have to consider the effect of children as well as the individual node being inserted.

1.8. Validating a Document Against a Content Model

The most obvious use for a content model (DTD or XML Schema or any Content Model) is to use it to validate that a given XML document is in fact a properly constructed instance of the document type described by this CM. This again falls into the document-editing world. The XML spec only discusses performing this test at the time the document is loaded into the "processor", which most of us have taken to mean that this check should be performed at parse time. But it is obviously desirable to be able to validate again a document -- or selected subtrees -- at other times. One such case would be validating an edited or newly constructed document before serializing it or otherwise passing it to other users. This issue also arises if the "internal subset" is altered -- or if the whole Content Model changes.

In the past, the DOM has allowed users to create invalid documents, and assumed the serializer would accept the task of detecting problems and announcing/repairing them when the document was written out in XML syntax... or that they would be checked for validity when read back in. We considered adding validity checks to the DOM's existing editing operations to prevent creation of invalid documents, but are currently inclined against this for several reasons. First, it would impose a significant amount of computational overhead to the DOM, which might be unnecessary in many situations, e.g., if the change is occurring in a context where we know the result will be valid. Second, "the shortest distance between two good documents may be through a bad document". Preventing a document from becoming temporarily invalid may impose a considerable amount of additional work on higher-level code and users Hence our current plan is to continue to permit editing to produce invalid DOMs, but provide operations which permit a user to check the validity of a node on demand.

Note that validation includes checking that ID attributes are unique, and that IDREFs point to IDs which actually exist.

1.9. Well-formedness Testing

XML defined the "well-formed" (*WF*) state for documents which are parsed without reference to their DTDs. Knowing that a document is well-formed may be useful by itself even when a DTD is available. For example, users may wish to deliberately save an invalid document, perhaps as a checkpoint before further editing. Hence, the CM feature will permit both full validity checking (see previous section) and "lightweight" WF checking, as requested by the caller, as well as processing entity declarations in the CM even if validation is not turned on. This falls within the document-editing world.

While the DOM inherently enforces some of XML's well-formedness conditions (proper nesting of elements, constraints on which children may be placed within each node), there are some checks that are not yet performed. These include:

- Character restrictions for text content and attribute values. Some characters aren't permitted even when expressed as numeric character entities
- The three-character sequence "]]>" in CDATASections.
- The two-character sequence "--" in comments. (Which, be it noted, some XML validators don't currently remember to test...)

In addition, Namespaces introduce their own concepts of well-formedness. Specifically:

- No two attributes on a single Element may have the same combination of namespaceURI and localName, even if their prefixes are different and hence they don't conflict under XML 1.0 rules.
- NamespaceURIs must be legal URI syntax. (Note that once we have this code, it may be reusable for the URI "datatype" in document content; see discussion of datatypes.)
- The mapping of namespace prefixes to their URIs must be declared and consistent. That isn't required during normal DOM operation, since we perform "early binding" and thereafter refer to nodes primarily via their namespaceURIs and localName. But it does become an issue when we want to serialize the DOM to XML syntax, and may be an issue if an application is assuming that all the declarations are present and correct. This may imply that we should provide a `namespaceNormalize` operation, which would create the implied declarations and reconcile conflicts in some reasonably standardized manner. This may be a major undertaking, since some DOMs may be using the namespace to direct subclassing of the nodes or similar special treatment; as with the existing `normalize` method, you may be left with a different-but-equivalent set of node objects.

In the past, the DOM has allowed users to create documents which violate these rules, and assumed the serializer would accept the task of detecting problems and announcing/repairing them when the document was written out in XML syntax. We considered adding WF checks to the DOM's existing editing operations to prevent WF violations from arising, but are currently inclined against this for two reasons.

First, it would impose a significant amount of computational overhead to the DOM, which might be unnecessary in many situations (for example, if the change is occurring in a context where we know the illegal characters have already been prevented from arising). Second, "the shortest distance between two good documents may be through a bad document" -- preventing a document from becoming temporarily ill-formed may impose a considerable amount of additional work on higher-level code and users. (Note possible issue for Serialization: In some applications, being able to save and reload marginally poorly-formed DOMs might be useful -- editor checkpoint files, for example.) Hence our current plan is to continue to permit editing to produce ill-formed DOMs, but provide operations which permit a user to check the well-formedness of a node on demand, and possibly provide some of the primitive (e.g., string-checking) functions directly.

2. Document Object Model Load and Save

Editors
Andy Heninger, IBM

2.1. Load and Save Requirements

DOM Level 3 will provide an API for loading XML source documents into a DOM representation and for saving a DOM representation as a XML document.

Some environments, such as the Java platform or COM, have their own ways to persist objects to streams and to restore them. There is no direct relationship between these mechanisms and the DOM load/save mechanism. This specification defines how to serialize documents only to and from XML format.

2.1.1. General Requirements

Requirements that apply to both loading and saving documents.

2.1.1.1. Document Sources

Documents must be able to be parsed from and saved to the following sources:

- Input and Output Streams
- URIs
- Files

Note that Input and Output streams take care of the in memory case. One point of caution is that a stream doesn't allow a base URI to be defined against which all relative URIs in the document are resolved.

2.1.1.2. Content Model Loading

While creating a new document using the DOM API, a mechanism must be provided to specify that the new document uses a pre-existing Content Model and to cause that Content Model to be loaded.

Note that while DOM Level 2 creation can specify a Content Model when creating a document (public and system IDs for the external subset, and a string for the subset), DOM Level 2 implementations do not process the Content Model's content. For DOM Level 3, the Content Model's content must be read.

2.1.1.3. Content Model Reuse

When processing a series of documents, all of which use the same Content Model, implementations should be able to reuse the already parsed and loaded Content Model rather than parsing it again for each new document.

This feature may not have an explicit DOM API associated with it, but it does require that nothing in this section, or the Content Model section, of this specification block it or make it difficult to implement.

2.1.1.4. Entity Resolution

Some means is required to allow applications to map public and system IDs to the correct document. This facility should provide sufficient capability to allow the implementation of catalogs, but providing catalogs themselves is not a requirement. In addition XML Base needs to be addressed.

2.1.1.5. Error Reporting

Loading a document can cause the generation of errors including:

- I/O Errors, such as the inability to find or open the specified document.
  XML well formedness errors.
  Validity errors

Saving a document can cause the generation of errors including:

- I/O Errors, such as the inability to write to a specified stream, URL, or file.
  Improper constructs, such as ’--’ in comments, in the DOM that cannot be represented as well formed XML.

This section, as well as the DOM Level 3 Content Model section should use a common error reporting mechanism. Well-formedness and validity checking are in the domain of the Content Model section, even though they may be commonly generated in response to an application asking that a document be loaded.

2.1.2. Load Requirements

The following requirements apply to loading documents.

2.1.2.1. Parser Properties and Options

Parsers may have properties or options that can be set by applications. Examples include:

- Expansion of entity references.
- Creation of entity ref nodes.
- Handling of white space in element content.
- Enabling of namespace handling.
- Enabling of content model validation.

A mechanism to set properties, query the state of properties, and to query the set of properties supported by a particular DOM implementation is required.

2.1.3. XML Writer Requirements

The fundamental requirement is to write a DOM document as XML source. All information to be serialized should be available via the normal DOM API.

2.1.3.1. XML Writer Properties and Options

There are several options that can be defined when saving an XML document. Some of these are:

- Saving to Canonical XML format.
- Pretty Printing.
- Specify the encoding in which a document is written.
- How and when to use character entities.
- Namespace prefix handling.
- Saving of Content Models.
- Handling of external entities.

2.1.3.2. Content Model Saving

Requirement from the Content Model group.

2.1.4. Other Items Under Consideration

The following items are not committed to, but are under consideration. Public feedback on these items is especially requested.

2.1.4.1. Incremental and/or Concurrent Parsing

Provide the ability for a thread that requested the loading of a document to continue execution without blocking while the document is being loaded. This would require some sort of notification or completion event when the loading process was done.

Provide the ability to examine the partial DOM representation before it has been fully loaded.

In one form, a document may be loaded asynchronously while a DOM based application is accessing the document. In another form, the application may explicitly ask for the next incremental portion of a document to be loaded.

2.1.4.2. Filtered Save

Provide the capability to write out only a part of a document. May be able to leverage TreeWalkers, or the Filters associated with TreeWalkers, or Ranges as a means of specifying the portion of the document to be written.

2.1.4.3. Document Fragments

Document fragments, as specified by the XML Fragment specification, should be able to be loaded. This is useful to applications that only need to process some part of a large document. Because the DOM is typically implemented as an in-memory representation of a document, fully loading large documents can require large amounts of memory.

XPath should also be considered as a way to identify XML Document fragments to load.

2.1.4.4. Document Fragments in Context of Existing DOM

Document fragments, as specified by the XML Fragment specification, should be able to be loaded into the context of an existing document at a point specified by a node position, or perhaps a range. This is a separate feature than simply loading document fragments as a new Node.

2.2. Issue List

2.2.1. Open Issues

Issue LS-Issue-10:
: Error Reporting. Loading will be reporting well-formedness and validation errors, just like CM. A common error reporting mechanism needs to be developed.

Issue LS-Issue-16:
: Loading and saving of content models - DTDs or Schemas - outside of the context of a document is not addressed.

Issue LS-Issue-17:
: Loading while validating using an already loaded content model is not addressed. Applications should be able to load a content model (issue 16), and then repeatedly reuse it during the loading of additional documents.

Issue LS-Issue-20:
: Action from September f2f to "add issues raised by schema discussion. What were these?

Issue LS-Issue-22:
: What do the bindings for things like InputStream look like in ECMA Script? Tentative resolution - InputStream will map to a binding dependent class or interface. For environments where nothing appropriate exists, a new interface will be created. This question is still being discussed.

Issue LS-Issue-27:
: How is validation during document loading handled when there are multiple possible content models associated with the document? How is one selected? The same question exists for documents in general, outside of the context of loading. Resolving the question for loading probably needs to wait until the more general question is understood.

Issue LS-Issue-32:
: Mimetypes. If the input being parsed is from http or something else that supplies types, and the type is something other than text/xml, should we parse it anyhow, or should we complain. Should there be an option?
: Tentative resolution: always parse, never complain. Reasons: 1. This is what all parsers do now, and no one has ever complained, at least not that I’m aware of. 2. Applications must have a pretty good

reason to suspect that they're getting xml or they wouldn't have invoked the parser. 3. All the test would do is to take something that might have worked (xml that is not known to the server) and turn it into an error. Non-xml is exceptionally unlikely to successfully parse (be well formed.)

Issue LS-Issue-34:

Features 2.1.4.1, 2 - XML Fragment Support. Should these be dropped? (Yes!)

Issue LS-Issue-35:

XPath based document load filter. It would be plausible to have a partial (filtered) document load based on selecting the portion of the document to load with an XPath expression. This facility could be in addition to the node-by-node filtering currently specified. Or we could drop the existing filter. Implementing an XPath based selective load would require that there be an XPath processor present in addition to the parser itself.

2.2.2. Resolved Issues

Issue LS-Issue-1:

Should these methods be in a new interface, or should they be added to the existing DOMImplementation Interface? I think that adding them to the existing interface is cleaner, because it helps avoid an explosion of new interfaces.

The methods are in a separate interface in this description for convenience in preparing the doc, so that I don't need to edit Core to add the methods. (The same argument could perhaps be made for implementations.)

Resolution: The methods are in a separate DOMImplementationLS interface. Because Load/Save is an optional module, we don't want to add its to the core DOMImplementation interface.

Issue LS-Issue-2:

SAX handles the setting of parser attributes differently. Rather than having distinct getters and setters for each attribute, it has a generic setter and getter of named properties, where properties are specified by a URL. This has an advantage in that implementations do not need to extend the interface when providing additional attributes.

If we choose to use strings, their syntax needs to be chosen. URIs would make sense, except for the fact that these are just names that do not refer to any resources. Dereferencing them would be meaningless. Yet the direction of the W3C is that all URIs must be dereferencable, and refer to something on the web.

Resolution: Use strings for properties. Use Java package name syntax for the identifying names. The question was revisited at the July f2f, with the same conclusion. But some discussion of using URLs continues.

This issue was revisited once again at the 9/2000 meeting. Now all DOM properties or features will be short, descriptive names, and we will recommend that all vendor-specific extensions be prefixed to avoid collisions, but will not make specific recommendations for the syntax of the prefix.

Issue LS-Issue-3:

It's not obvious what name to choose for the parser interface. Taking any of the names already in use by parser implementations would create problems when trying to support both the new API and the existing old API. That leaves out `DocumentBuilder` (Sun) and `DOMParser` (Xerces).

Resolution: This is issue really just a comment. The "resolution" is in the names appearing in the API.

Issue LS-Issue-4:

Question: should ResolveEntity pass a baseURI string back to the application, in addition to the publicId, systemId, and/or stream? Particularly in the case of an input stream.

Resolution: No. Sax2 explicitly says that the system ID URI must be fully resolved before passing it out to the entity resolve. We will follow SAX's lead on this unless some additional use case surfaces. This is from the 9/2000 f2f, and reverses an earlier decision.

Issue LS-Issue-5:

When parsing a document that contains errors, should the whole document be decreed unusable, or should we say that portions prior to the point where the error was detected are OK?

Resolution: In the case of errors in the XML source, what, if any, document is returned is implementation dependent.

Issue LS-Issue-6:

The relationship between SAXExceptions and DOM exceptions seems confusing.

Resolution: This issue goes away because we are no longer using SAX. Any exceptions will be DOM Exceptions.

Issue LS-Issue-7:

Question: In the original Java definition, are the strings returned from the methods `SAXException.toString()` and `SAXException.getMessage()` always the same? If not, we need to add another attribute.

Resolution: No longer an issue because we are no longer using SAX.

Issue LS-Issue-8:

JAXP defines a mechanism, based on Java system properties, by which the Document Builder Factory locates the specific parser implementation to be used. This ability to redirect to different parsers is a key feature of JAXP. How this redirection works in the context of this design may be something that needs to be defined separately for each language binding.

This question was discussed at the July f2f, without resolution. Agreed that the feature is not critical to the rest of the API, and can be postponed.

Resolution: The issue is moving to core, where it is part of the bigger question of where does the DOM implementation come from, and how do multiple implementations coexist. Allowing separate, or mix-and-match, specification of the parser and the rest of the DOM is not generally practical because parsers generally have some degree of private knowledge about their DOMs.

Issue LS-Issue-9:

The use of interfaces from SAX2 raises some questions. The Java bindings for these interfaces need to be exactly the SAX2 definitions, including the original org.xml.sax package name.

The IDL presented here for these interfaces is an attempt to map the Java into IDL, but it will certainly not round-trip accurately - Java bindings generated from the IDL will not match the original Java.

The reasons for using the SAX interfaces are that they are well designed, widely implemented and used, and provide what is needed. Designing something new would create confusion for application developers (which should be used?) and make extra work for implementers of the DOM, most of whom probably already provide SAX, all for no real gain.

Resolution: Problem is gone. We are not using SAX2. The design will borrow features and concepts from SAX2 when it makes sense to do so.

Issue LS-Issue-11:

Another Error Reporting Question. We decided at the June f2f that validity errors should not be exceptions. This means that a document load operation could encounter multiple errors. Should these

be collected and delivered as some sort of collection at the (otherwise) successful completion of the load, or should there be some sort of callback? Callbacks are harder for applications to deal with.
Resolution: Provide a callback mechanism. Provide a default error handler that throws an exception and stops further processing. From July f2f.

Issue LS-Issue-12:
Definition of "Non-validating". Exactly how much processing is done by "non-validating" parsers is not fully defined by the XML specification. In particular, they are not required to read any external entities, but are not prohibited from doing so.
Another common user request: a mode that completely ignores DTDs, both and external. Such a parser would not conform to XML 1.0, however.
For the documents produced by a non-validating load to be the same, we need to tie down exactly what processing must be done. The XML Core WG also has question as an open issue .
Some discussion is at http://lists.w3.org/Archives/Member/w3c-xml-core-wg/2000JanMar/0192.html
Here is proposal: Have three classes of parsers

- Minimal. No external entities of any type are accessed. DTD subset is processes normally, as required by XML 1.0, including all entity definitions it contains.
- Non-Validating. All external entities are read. Does everything except validation.
- Validating. As defined by XML 1.0 rec.

Resolution: Use the options from SAX2. These provide separate flags for validation, reading of external general entities and reading of external parameter entities.

Issue LS-Issue-13:
Use of System or Language specific types for Input and Output
Loading and Saving requires that one of the possible sources or destinations of the XML data be some sort of stream that can be used with io streams or memory buffers, or anything else that might take or supply data. The type will vary, depending on the language binding.
The question is, what should be put into the IDL interfaces for these? Should we define an XML stream to abstract out the dependency, or use system classes directly in the bindings?
Resolution: Define IDL types for use in the rest of the interface definitions. These types will be mapped directly to system types for each language binding

Issue LS-Issue-14:
Should there be separate DOM modules for browser or scripting style loading (document.load("whatever")) and server style parsers? It's probably easy for the server style parsers to implement the browser style interface, but the reverse may not be true.
Resolution: Yes. A client application style API will be provided.

Issue LS-Issue-15:
System Exceptions. Loading involves file opens and reads, and these can result in a variety of system errors that may already have associated system exceptions. Should these system exceptions pass through as is, or should they be some how wrapped in DOMExceptions, or should there be a parallel set DOM Exceptions, or what?
Resolution: Introduce a new DOMSystemException to standardize the reporting of common I/O errors across different DOM environments. Let it wrap an underlying system exception or error code when appropriate. To be defined in the common ErrorReporting module, to be shared with ContentModel.

Issue LS-Issue-18:
For the list of parser properties, which must all implementations recognize, which settings must all implementations support, and which are optional?

Resolution: Done

Issue LS-Issue-19:

DOMOutputStream: should this be an interface with methods, or just an opaque type that maps onto an appropriate binding-specific stream type?

If we specify an actual interface with methods, applications can implement it to wrap any arbitrary destination that they may have. If we go with the system type it's simpler to output to that type of stream, but harder otherwise.

Resolution: Opaque.

Issue LS-Issue-21:

Define exceptions. A `DOMSystemException` needs to be defined as part of the error handling module that is to be shared with CM. Common I/O type errors need to be defined for it, so that they can be reported in a uniform way. A way to embed errors or exceptions from the OS or language environment is needed, to provide full information to applications that want it.

Resolution: Duplicate of issue #15

Issue LS-Issue-23:

To Do: Add a method or methods to DOMBuilder that will provide information about a parser feature - is the name recognized, which (boolean) values are supported - without throwing exceptions.

Resolution: Done. Added canSetFeature.

Issue LS-Issue-24:

Clearly identify which of the parser properties must be recognized, and which of their settings must be supported by all conforming implementations.

Resolution: Done. All must be recognized.

Issue LS-Issue-25:

How does the validation property work in SAX, and how should it work for us? The default value in SAX2 is "true". Non-validating parsers only support a value of false. Does this mean that the default depends on the parser, or that some sort of an error happens if a parse is attempted before resetting the property, or what?

The same question applies to the External Entities properties too.

Resolution: Make the default value for the validation property be false.

Issue LS-Issue-26:

Do we want to rename the "auto-validation" property to "validate-if-cm"? Proposed at f2f. Resolution unclear.

Resolution: Changed the name to "validate-if-cm".

Issue LS-Issue-29:

Should all properties except namespaces default to false? Discussed at f2f. I'm not so sure now. Some of the properties have somewhat non-standard behavior when false - leaving out ER nodes or whitespace, for example - and support of false will probably not even be required.

Resolution: Not all properties should default to false. But validation should.

Issue LS-Issue-28:

To do: add new parser property "createEntityNodes". default is true. Illegal for it to be false and createEntityReferenceNodes to be true.

Is this really what we want?

Resolution: new feature added.

Issue LS-Issue-30:

Possible additional parser features - option to not create CDATA nodes, and to merge CDATA

contents with adjacent TEXT nodes if they exist. Otherwise just create a TEXT node.
Option to omit Comments.
Resolution: new feature added.

Issue LS-Issue-31:
We now have an option for fixing up name space declarations and prefixes on serialization. Should we specify how this is done, so that the documents from different implementations of serialization will use the same declarations and prefixes, or should we leave the details up to the implementation?
Resolution: The exact form of the name space fixup is implementation dependent. The only requirement is that all elements and attributes end up with the correct name space URI.

Issue LS-Issue-33:
Unicode Character Normalization Problems. It turns out that for some code pages, normalizing a Unicode representation, translating to the code page, then translating back to Unicode can result in un-normalized Unicode. Mark Davis says that this can happen with Vietnamese and maybe with Hebrew.
This means that the suggested W3C model of normalization on serialization (early normalization) may not work, and that the receiver of the data may need to normalize it again, just in case.
Resolution: The scenario described is a quality-of-implementation issue. A transcoder converting from the one of the troublesome code pages to a Unicode representation should be responsible for re-normalizing the output.

2.3. Interfaces

This section defines an API for loading (parsing) XML source documents into a DOM representation and for saving (serializing) a DOM representation as an XML document.

The proposal for loading is influenced by Sun's JAXP API for XML Parsing in Java, http://java.sun.com/xml/download.html, and by SAX2, available at http://www.megginson.com/SAX/index.html

2.3.1. Interface Summary

Here is a list of each of the interfaces involved with the Loading and Saving XML documents.

- `DOMImplementationLS` [p.56] -- A new `DOMImplementation` interface that provides the factory methods for creating the objects required for loading and saving.
- `DOMBuilder` [p.56] -- A parser interface.
- `DOMInputSource` [p.62] -- Encapsulate information about the source of the XML to be loaded.
- `DOMEntityResolver` [p.63] -- During loading, provides a way for applications to redirect references to external entities.
- `DOMBuilderFilter` [p.65] -- Provide the ability to examine and optionally remove Element nodes as they are being processed during the parsing of a document.
- `DOMWriter` [p.66] -- An interface for writing out or serializing DOM documents.
- `DocumentLS` [p.70] -- Provides a client or browser style interface for loading and saving.
- `ParserErrorEvent` [p.71] -- ParserErrorEvent is the event that is fired if there's an error in the XML document being parsed using the methods of DocumentLS.

2.3.2. Interfaces

Interface *DOMImplementationLS*

`DOMImplementationLS` contains the factory methods for creating objects implementing the `DOMBuilder` [p.56] (parser) and `DOMWriter` [p.66] interfaces.

An object implementing DOMImplementationLS is obtained by casting from DOMImplementation to DOMImplementationLS, using the customary casting facilities from the programming language in use. Implementations supporting the Load and Save feature must implement the DOMImplementationLS interface on whatever object implements the DOMImplementation interface.

IDL Definition

```
interface DOMImplementationLS {
  DOMBuilder         createDOMBuilder();
  DOMWriter          createDOMWriter();
};
```

Methods

`createDOMBuilder`

Create a new `DOMBuilder` [p.56] . The newly constructed parser may then be configured by means of its `setFeature()` method, and used to parse documents by means of its `parse()` method.

Return Value

`DOMBuilder` [p.56]	The newly created parser object.

No Parameters

No Exceptions

`createDOMWriter`

Create a new `DOMWriter` [p.66] object. `DOMWriters` are used to serialize a DOM tree back into source XML form.

Return Value

`DOMWriter` [p.66]	The newly created `DOMWriter` object.

No Parameters

No Exceptions

Interface *DOMBuilder*

A parser interface.

`DOMBuilder` provides an API for parsing XML documents and building the corresponding DOM document tree. A `DOMBuilder` instance is obtained from the `DOMImplementationLS` [p.56] interface by invoking its `createDOMBuilder()` method.

`DOMBuilders` have a number of named properties that can be queried or set. Here is a list of properties that must be recognized by all implementations.

- **namespaces**
  true: perform Namespace processing.
  false: do not perform name space processing.
  default: true.
  supported values: true: required; false: optional
- **namespace-declarations**
  true: include namespace declarations (xmlns attributes) in the DOM document.
  false: discard all namespace declarations. In either case, namespace prefixes will be retained.
  default: true.
  supported values: true: required; false: optional
- **validation**
  true: report validation errors (setting true also will force the external-general-entities and external-parameter-entities properties to be set true.) Also note that the `validate-if-cm` feature will alter the validation behavior when this feature is set true.
  false: do not report validation errors.
  default: false.
  supported values: true: optional; false: required
- **external-general-entities**
  true: include all external general (text) entities.
  false: do not include external general entities.
  default: true.
  supported values: true: required; false: optional
- **external-parameter-entities**
  true: include all external parameter entities.
  false: do not include external parameter entities.
  default: true.
  supported values: true: required; false: optional
- **validate-if-cm**
  true: when both this feature and validation are true, enable validation only when the document being processed has a content model. Documents without content models are parsed without validation.
  false: the validation feature alone controls whether the document is checked for validity. Documents without content models are not valid.
  default: false.
  supported values: true: optional; false: required
- **create-entity-ref-nodes**
  true: create entity reference nodes in the DOM document. Setting this value true will also set create-entity-nodes to be true
  false: omit all entity reference nodes from the DOM document, putting the entity expansions directly in their place.
  default: true.
  supported values: true: required; false: optional
- **entity-nodes**

true: create entity nodes in the DOM document.
false: omit all entity nodes from the DOM document. Setting this value false will also set create-entity-ref-nodes false.
default: true.
supported values: true: required; false: optional

- **white-space-in-element-content**
  true: include white space in element content in the DOM document. This is sometimes referred to as ignorable white space
  false: omit said white space. Note that white space in element content will only be omitted if it can be identified as such, and not all parsers may be able to do so.
  default: true.
  supported values: true: required; false: optional
- **cdata-nodes**
  true: Create DOM CDATA nodes in response to the appearance of CDATA sections in the source XML.
  false: Do not create CDATA nodes in the DOM document. The content of any CDATA sections in the source XML appears in the DOM as if it had been normal (non-CDATA) content. If a CDATA section is adjacent to other content, the combined content appears in a single TEXT node. The DOM Document produced by the DOMBuilder will not have adjacent TEXT nodes.
  default: true
  supported values: false: optional; true: required
- **comments**
  true: Include XML comments in the DOM document
  false: Discard XML comments, do not create Comment nodes in the DOM Document resulting from a parse.
  default: true
  supported values: false: required; true: required
- **charset-overrides-xml-encoding**
  true: If a higher level protocol such as http provides an indication of the character encoding of the input stream being processed, that will override any encoding specified in the XML or TEXT declaration of the XML. Explicitly setting an encoding in the DOMInputSource overrides encodings from the protocol.
  false: Any character set encoding information from higher level protocols is ignored by the parser.
  default: true
  supported values: false: required; true: required

IDL Definition

```
interface DOMBuilder {
           attribute DOMEntityResolver  entityResolver;
           attribute DOMErrorHandler  errorHandler;
           attribute DOMBuilderFilter  filter;
  void               setFeature(in DOMString name,
                                in boolean state)
                                        raises(DOMException);
  boolean            supportsFeature(in DOMString name);
  boolean            canSetFeature(in DOMString name,
                                   in boolean state);
```

```
  boolean            getFeature(in DOMString name)
                                   raises(DOMException);
  Document           parseURI(in DOMString uri)
                                   raises(DOMException,
                                          DOMSystemException);
  Document           parseDOMInputSource(in DOMInputSource is)
                                   raises(DOMException,
                                          DOMSystemException);
};
```

Attributes

`entityResolver` of type `DOMEntityResolver` [p.63]

If a `DOMEntityResolver` [p.63] has been specified, each time a reference to an external entity is encountered the `DOMBuilder` will pass the public and system IDs to the entity resolver, which can then specify the actual source of the entity.

`errorHandler` of type `DOMErrorHandler` [p.40]

In the event that an error is encountered in the XML document being parsed, the `DOMDcoumentBuilder` will call back to the `errorHandler` with the error information.

Note: The DOMErrorHandler interface is being developed separately, in conjunction with the design of the content model and validation module.

`filter` of type `DOMBuilderFilter` [p.65]

When the application provides a filter, the parser will call out to the filter at the completion of the construction of each `Element` node. The filter implementation can choose to remove the element from the document being constructed or to terminate the parse early.

Methods

`canSetFeature`

query whether setting a feature is supported.
The feature name has the same form as a DOM hasFeature string.
It is possible for a `DOMBuilder` to recognize a feature name but to be unable to set its value.

Parameters

`name` of type `DOMString`

The feature name, which is a DOM has-feature style string.

`state` of type `boolean`

The requested state of the feature (true or false).

Return Value

`boolean`	true if the feature could be successfully set to the specified value, or false if the feature is not recognized or the requested value is not supported. The value of the feature itself is not changed.

No Exceptions

`getFeature`

Look up the value of a feature.
The feature name has the same form as a DOM hasFeature string

Parameters
`name` of type `DOMString`
The feature name, which is a string with DOM has-feature syntax.
Return Value

`boolean`	The current state of the feature (true or false).

Exceptions

`DOMException`	Raise a NOT_FOUND_ERR When the `DOMBuilder` does not recognize the feature name.

`parseDOMInputSource`
Parse an XML document from a location identified by an `DOMInputSource` [p.62] .
Parameters
`is` of type `DOMInputSource` [p.62]
The `DOMInputSource` from which the source document is to be read.
Return Value

`Document` [p.26]	The newly created and populated`Document`.

Exceptions

`DOMException`	Exceptions raised by `parseDOMInputSource()` originate with the installed ErrorHandler, and thus depend on the implementation of the `DOMErrorHandler` [p.40] interfaces. The default ErrorHandlers will raise a `DOMException` if any form of XML validation or well formedness error or warning occurs during the parse, but application defined errorHandlers are not required to do so.
`DOMSystemException`	Exceptions raised by `parseDOMInputSource()` originate with the installed ErrorHandler, and thus depend on the implementation of the `DOMErrorHandler` [p.40] interfaces. The default ErrorHandlers will raise a `DOMSystemException` if any form I/O or other system error occurs during the parse, but application defined ErrorHandlers are not required to do so.

`parseURI`
Parse an XML document from a location identified by an URI.
Parameters

`uri` of type `DOMString`
The location of the XML document to be read.

Return Value

`Document` [p.26]	The newly created and populated`Document`.

Exceptions

`DOMException`	Exceptions raised by `parseURI()` originate with the installed ErrorHandler, and thus depend on the implementation of the `DOMErrorHandler` [p.40] interfaces. The default error handlers will raise a `DOMException` if any form of XML validation or well formedness error or warning occurs during the parse, but application defined errorHandlers are not required to do so.
`DOMSystemException`	Exceptions raised by `parseURI()` originate with the installed ErrorHandler, and thus depend on the implementation of the `DOMErrorHandler` [p.40] interfaces. The default error handlers will raise a DOMSystemException if any form I/O or other system error occurs during the parse, but application defined error handlers are not required to do so.

`setFeature`
Set the state of a feature.
The feature name has the same form as a DOM hasFeature string.
It is possible for a `DOMBuilder` to recognize a feature name but to be unable to set its value.

Parameters

`name` of type `DOMString`
The feature name, which is a DOM has-feature style string.
`state` of type `boolean`
The requested state of the feature (true or false).

Exceptions

`DOMException`	Raise a NOT_SUPPORTED_ERR exception When the `DOMBuilder` recognizes the feature name but cannot set the requested value. Raise a NOT_FOUND_ERR When the `DOMBuilder` does not recognize the feature name.

No Return Value

`supportsFeature`

query whether the `DOMBuilder` recognizes a feature name.
The feature name has the same form as a DOM hasFeature string.
It is possible for a `DOMBuilder` to recognize a feature name but to be unable to set its value. For example, a non-validating parser would recognize the feature "validation", would report that its value was false, and would raise an exception if an attempt was made to enable validation by setting the feature to true.

Parameters

`name` of type `DOMString`

The feature name, which has the same syntax as a DOM has-feature string.

Return Value

`boolean`	true if the feature name is recognized by the `DOMBuilder`. False if the feature name is not recognized.

No Exceptions

Interface *DOMInputSource*

This interface represents a single input source for an XML entity.

This interface allows an application to encapsulate information about an input source in a single object, which may include a public identifier, a system identifier, a byte stream (possibly with a specified encoding), and/or a character stream.

The exact definitions of a byte stream and a character stream are binding dependent.

There are two places that the application will deliver this input source to the parser: as the argument to the `parseDOMInputSource` method, or as the return value of the `DOMEntityResolver.resolveEntity` [p.64] method.

The `DOMBuilder` [p.56] will use the `DOMInputSource` object to determine how to read XML input. If there is a character stream available, the parser will read that stream directly; if not, the parser will use a byte stream, if available; if neither a character stream nor a byte stream is available, the parser will attempt to open a URI connection to the resource identified by the system identifier.

An `DOMInputSource` object belongs to the application: the parser shall never modify it in any way (it may modify a copy if necessary).

IDL Definition

```
interface DOMInputSource {
          attribute DOMInputStream   byteStream;
          attribute DOMReader        characterStream;
          attribute DOMString        encoding;
          attribute DOMString        publicId;
          attribute DOMString        systemId;
};
```

Attributes

`byteStream` of type `DOMInputStream`
: An attribute of a language-binding dependent type that represents a stream of bytes.
The parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.
If the application knows the character encoding of the byte stream, it should set the encoding property. Setting the encoding in this way will override any encoding specified in the XML declaration itself.

`characterStream` of type `DOMReader`
: An attribute of a language-binding dependent type that represents a stream of 16 bit values (utf-16 encoded characters).
If a character stream is specified, the parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

`encoding` of type `DOMString`
: The character encoding, if known. The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML 1.0 recommendation).
This attribute has no effect when the application provides a character stream. For other sources of input, an encoding specified by means of this attribute will override any encoding specified in the XML or text declaration of the XML, or an encoding obtained from a higher level protocol, such as http.

`publicId` of type `DOMString`
: The public identifier for this input source. The public identifier is always optional: if the application writer includes one, it will be provided as part of the location information.

`systemId` of type `DOMString`
: The system identifier for this input source. The system identifier is optional if there is a byte stream or a character stream, but it is still useful to provide one, since the application can use it to resolve relative URIs and can include it in error messages and warnings (the parser will attempt to open a connection to the URI only if there is no byte stream or character stream specified).
If the application knows the character encoding of the object pointed to by the system identifier, it can register the encoding by setting the encoding attribute.
If the system ID is a URL, it must be fully resolved.

Interface *DOMEntityResolver*

`DOMEntityResolver` Provides a way for applications to redirect references to external entities.

Applications needing to implement customized handling for external entities must implement this interface and register their implementation by setting the `entityResolver` property of the `DOMBuilder` [p.56] .

The `DOMBuilder` [p.56] will then allow the application to intercept any external entities (including the external DTD subset and external parameter entities) before including them.

Many DOM applications will not need to implement this interface, but it will be especially useful for applications that build XML documents from databases or other specialized input sources, or for applications that use URI types other than URLs.

DOMEtityResolver is based on the SAX2 EntityResolver interface, described at http://www.megginson.com/SAX/Java/javadoc/org/xml/sax/EntityResolver.html

IDL Definition

```
interface DOMEntityResolver {
  DOMInputSource     resolveEntity(in DOMString publicId,
                                   in DOMString systemId )
                                        raises(DOMSystemException);
};
```

Methods

resolveEntity

Allow the application to resolve external entities.
The DOMBuilder [p.56] will call this method before opening any external entity except the top-level document entity (including the external DTD subset, external entities referenced within the DTD, and external entities referenced within the document element); the application may request that the DOMBuilder resolve the entity itself, that it use an alternative URI, or that it use an entirely different input source.
Application writers can use this method to redirect external system identifiers to secure and/or local URIs, to look up public identifiers in a catalogue, or to read an entity from a database or other input source (including, for example, a dialog box).
If the system identifier is a URL, the DOMBuilder [p.56] must resolve it fully before reporting it to the application through this interface.

Note: See issue #4. An alternative would be to pass the URL out without resolving it, and to provide a base as an additional parameter. SAX resolves URLs first, and does not provide a base.

Parameters

publicId of type DOMString
The public identifier of the external entity being referenced, or null if none was supplied.

systemId of type DOMString
The system identifier of the external entity being referenced.

Return Value

DOMInputSource [p.62]	A DOMInputSource object describing the new input source, or null to request that the parser open a regular URI connection to the system identifier.

Exceptions

DOMSystemException	Any DOMSystemException, possibly wrapping another exception.

Interface *DOMBuilderFilter*

`DOMBuilderFilters` provide applications the ability to examine Element nodes as they are being constructed during a parse. As each elements is examined, it may be modified or removed, or the entire parse may be terminated early.

At the time any of the filter methods are called by the parser, the owner Document and DOMImplementation objects exist and are accessible.

All validity checking while reading a document occurs on the source document as it appears on the input stream, not on the DOM document as it is built in memory. With filters, the document in memory may be a subset of the document on the stream, and its validity may have been affected by the filtering.

IDL Definition

```
interface DOMBuilderFilter {
  boolean            startElement(in Element element);
  boolean            endElement(in Element element);
};
```

Methods

`endElement`

This method will be called by the parser at the completion of the parse of each element. The element node will exist and be complete, as will all of its children, and their children, recursively. The element's parent node will also exist, although that node may be incomplete, as it may have additional children that have not yet been parsed.
From within this method, the new node may be freely modified - children may be added or removed, text nodes modified, etc. This node may also be removed from its parent node, which will prevent it from appearing in the final document at the completion of the parse. Aside from this one operation on the node's parent, the state of the rest of the document outside of this node is not defined, and the affect of any attempt to navigate to or modify any other part of the document is undefined.
For validating parsers, the checks are made on the original document, before any modification by the filter. No validity checks are made on any document modifications made by the filter.

Parameters

`element` of type `Element`

The newly constructed element. At the time this method is called, the element is complete - it has all of its children (and their children, recursively) and attributes, and is attached as a child to its parent.

Return Value

`boolean`	return true

No Exceptions

`startElement`

This method will be called by the parser after each Element start tag has been scanned, but before the remainder of the Element is processed. The intent is to allow the element, including any children, to be efficiently skipped.

The element node passed to `startElement` for filtering will include all of the Element's attributes, but none of the children nodes. The Element may not yet be in place in the document being constructed (it may not have a parent node.)

A StartElement filter function may access or change the attributers for the Element. Changing NameSpace declarations will have no effect on name space resolution by the parser.

For efficiency, the Element node passed to the filter may not be the same one as is actually placed in the tree if the node is accepted. And the actual node (node object identity) may be reused during the process of reading in and filtering a document.

Parameters

`element` of type `Element`

The newly encountered element. At the time this method is called, the element is incomplete - it will have its attributes, but no children.

Return Value

`boolean`	return true if this Element should be included in the DOM document being built. Return false if the Element and all of its children should be skipped.

No Exceptions

Interface *DOMWriter*

DOMWriter provides an API for serializing (writing) a DOM document out in the form of a source XML document. The XML data is written to an output stream, the type of which depends on the specific language bindings in use.

Three options are available for the general appearance of the formatted output: As-is, canonical and reformatted.

- As-is formatting leaves all "white space in element content" and new-lines unchanged. If the DOM document originated as XML source, and if all white space was retained, this option will come the closest to recovering the format of the original document. (There may still be differences due to normalization of attribute values and new-line characters or the handling of character references.)
- Canonical formatting writes the document according to the rules specified by W3C Canonical XML Version 1.0. http://www.w3.org/TR/xml-c14n
- Reformatted output has white space and newlines adjusted to produce a pretty-printed, indented, human-readable form. The exact form of the transformations is not specified.

`DOMWriter` accepts any node type for serialization. For nodes of type `Document` [p.26] or `Entity`, well formed XML will be created. The serialized output for these node types is either as a Document or an External Entity, respectively, and is acceptable input for an XML parser. For all other types of nodes the serialized form is not specified, but should be something useful to a human for debugging or diagnostic purposes. Note: rigorously designing an external (source) form for stand-alone node types that don't already have one defined by the XML rec seems a bit much to take on here.

Within a Document or Entity being serialized, Nodes are processed as follows

- Documents are written including an XML declaration and a DTD subset, if one exists in the DOM. Writing a document node serializes the entire document.
- Entity nodes, when written directly by `DOMWriter.writeNode()`, output a Text Decl and the entity expansion. The resulting output will be valid as an external entity.
  No output is generated for any entity nodes when writing a `Document` [p.26] .
- Entity References nodes are serializes as an entity reference of the form `"&entityName;"`) in the output. Child nodes (the expansion) of the entity reference are ignored.
- CDATA sections containing content characters that can not be represented in the specified output encoding are handled according to the "split-cdata-sections" option.
  If the option is true, CDATA sections are split, and the unrepresentable characters are serialized as numeric character references in ordinary content. The exact position and number of splits is not specified.
  If the option is false, unrepresentable characters in a CDATA section are reported as errors. The error is not recoverable - there is no mechanism for supplying alternative characters and continuing with the serialization.
- All other node types (Element, Text, etc.) are serialized to their corresponding XML source form.

Within the character data of a document (outside of markup), any characters that cannot be represented directly are replaced with character references. Occurrences of '<' and '&' are replaced by the predefined entities < and &amp. The other predefined entities (&gt, &apos, etc.) are not used; these characters can be included directly. Any character that can not be represented directly in the output character encoding is serialized as a numeric character reference.

Attributes not containing quotes are serialized in quotes. Attributes containing quotes but no apostrophes are serialized in apostrophes (single quotes). Attributes containing both forms of quotes are serialized in quotes, with quotes within the value represented by the predefined entity ". Any character that can not be represented directly in the output character encoding is serialized as a numeric character reference.

Within markup, but outside of attributes, any occurrence of a character that cannot be represented in the output character encoding is reported as an error. An example would be serializing the element <LaCañada/> with the encoding=US-ASCII

Unicode Character Normalization. When requested by setting the `normalizeCharacters` option on DOMWriter, all data to be serialized, both markup and character data, is normalized according to the rules defined by Unicode Canonical Composition, Normalization Form C. The normalization process affects only the data as it is being written; it does not alter the DOM's view of the document after serialization has completed. The W3C character model and normalization are described at http://www.w3.org/TR/charmod/#TextNormalization. Unicode normalization forms are described at http://www.unicode.org/unicode/reports/tr15/

Name space checking and fixup during serialization is a user option. When the option is selected, the serialization process will verify that name space declarations, name space prefixes and the name space URIs associated with Elements and Attributes are consistent. If inconsistencies are found, the serialized form of the document will be altered to remove them. The exact form of the alterations are not defined, and are implementation dependent.

Any changes made affect only the name space prefixes and declarations appearing in the serialized data. The DOM's view of the document is not altered by the serialization operation, and does not reflect any changes made to name space declarations or prefixes in the serialized output.

`DOMWriters` have a number of named properties that can be queried or set. Here is a list of properties that must be recognized by all implementations.

- **normalizeCharacters**
  true: Perform Unicode Normalization of the characters in document as they are written out. Only the characters being written are (potentially) altered. The DOM document itself is unchanged.
  false: do not perform character normalization.
  default: true.
  supported values: true: required; false: required.
- **namespaceFixup**
  true: Check namespace declarations and prefixes for consistency, and fix them in the serialized data if they are inconsistent.
  false: Perform no special checks on name space declarations, prefixes or URIs.
  default: true;
  supported values: true: required; false: required.
- **split-cdata-sections**
  true: Split CDATA sections containing characters that can not be represented in the output encoding, and output the characters using numeric character references.
  false: Signal an error if a CDATA section contains an unrepresentable character.
  supported values: true: required; false: required.

IDL Definition

```
interface DOMWriter {
           attribute DOMString        encoding;
  readonly attribute DOMString        lastEncoding;
           attribute unsigned short   format;
  // Modified in DOM Level 3:
           attribute DOMString        newLine;
  void               writeNode(in DOMOutputStream destination,
                               in Node node)
                                        raises(DOMSystemException);
};
```

Attributes

`encoding` of type `DOMString`
The character encoding in which the output will be written.
The encoding to use when writing is determined as follows:
- If the encoding attribute has been set, that value will be used.
- If the encoding attribute is null or empty, but the item to be written includes an encoding declaration, that value will be used.
- If neither of the above provides an encoding name, a default encoding of "utf-8" will be used.

The default value is null.

`format` of type `unsigned short`
As-is, canonical or reformatted. *Need to add constants for these.*
The default value is as-is.

`lastEncoding` of type `DOMString`, readonly
The actual character encoding that was last used by this formatter. This convenience method allows the encoding that was used when serializing a document to be directly obtained.

`newLine` of type `DOMString`, modified in **DOM Level 3**
The end-of-line character(s) to be used in the XML being written out. The only permitted values are these:
- null: Use a default end-of-line sequence. DOM implementations should choose the default to match the usual convention for text files in the environment being used. Implementations must choose a default sequence that matches one of those allowed by the XML Recommendation, http://www.w3.org/TR/REC-xml#sec-line-ends
- CR
- CR-LF
- LF

The default value for this attribute is null.

Methods

`writeNode`
Write out the specified node as described above in the description of `DOMWriter`. Writing a Document or Entity node produces a serialized form that is well formed XML. Writing other node types produces a fragment of text in a form that is not fully defined by this document, but that should be useful to a human for debugging or diagnostic purposes.

Parameters

`destination` of type `DOMOutputStream`
The destination for the data to be written.

`node` of type `Node`
The `Document` [p.26] or `Entity` node to be written. For other node types, something sensible should be written, but the exact serialized form is not specified.

Exceptions

`DOMSystemException`	This exception will be raised in response to any sort of IO or system error that occurs while writing to the destination. It may wrap an underlying system exception.

No Return Value

Interface *DocumentLS*

The DocumentLS interface provides a mechanism by which the content of a document can be replaced with the DOM tree produced when loading a URL, or parsing a string. The expectation is that an instance of the DocumentLS interface can be obtained by using binding-specific casting methods on an instance of the Document interface.

IDL Definition

```
interface DocumentLS {
           attribute boolean          async;
  void               abort();
  boolean            load(in DOMString url);
  boolean            loadXML(in DOMString source);
  DOMString          saveXML(in Node node)
                                        raises(DOMException);
};
```

Attributes

`async` of type `boolean`

Indicates whether the method load should be synchronous or asynchronous. When the async attribute is set to true the load method returns control to the caller before the document has completed loading. The default value of this property is true.
Setting the value of this attribute might throw NOT_SUPPORTED_ERR if the implementation doesn't support the mode the attribute is being set to.
<< ISSUE >> Should the DOM spec define the default value of this property? What if implementing both async and sync IO is impractical in some systems?

Methods

`abort`

If the document is currently being loaded as a result of the method `load()` being invoked the loading and parsing is immediately aborted. The possibly partial result of parsing the document is discarded and the document is cleared.

No Parameters

No Return Value

No Exceptions

`load`

Replaces the content of the document with the result of parsing the given URL. Invoking this method will either block the caller or return to the caller immediately depending on the value of the async attribute. Once the document is fully loaded the document will fire a "load" event that the caller can register as a listener for. If an error occurs the document will fire an "error" event so that the caller knows that the load failed (see `ParserErrorEvent` [p.71]).

Parameters

`url` of type `DOMString`

The URL for the XML file to be loaded.

Return Value

`boolean`	If async is set to true load() returns true if the document load was successfully initiated. If an error occurred when initiating the document load load() returns false. If async is set to false load() returns true if the document was successfully loaded and parsed. If an error occurred when either loading or parsing the URL load() returns false.

No Exceptions

`loadXML`

Replace the content of the document with the result of parsing the input string, this method is always synchronous.

Parameters

`source` of type `DOMString`

A string containing an XML document.

Return Value

`boolean`	True if parsing the input string succeeded without errors, otherwise false.

No Exceptions

`saveXML`

Save the document or the given node to a string (i.e. serialize the document or node).

Parameters

`node` of type `Node`

Specifies what to serialize, if this parameter is null the whole document is serialized, if it's non-null the given node is serialized.

Return Value

`DOMString`	The serialized document or node.

Exceptions

`DOMException`	DOMException WRONG_DOCUMENT_ERR: Raised if the node passed in as the node parameter is from an other document.

Interface *ParserErrorEvent*

ParserErrorEvent is the event that is fired if there's an error in the XML document being parsed.

IDL Definition

```
interface ParserErrorEvent {
  readonly attribute long            errorCode;
  readonly attribute long            filepos;
  readonly attribute long            line;
  readonly attribute long            linepos;
  readonly attribute DOMString       reason;
  readonly attribute DOMString       srcText;
  readonly attribute DOMString       url;
};
```

Attributes

`errorCode` of type `long`, readonly
: An non-zero implementation dependent error code describing the error, or 0 if there is no error.

`filepos` of type `long`, readonly
: The byte position in the file where the error occurred.

`line` of type `long`, readonly
: The line number where the error occurred.

`linepos` of type `long`, readonly
: The number of the character on the line where the error occurred.

`reason` of type `DOMString`, readonly
: An implementation dependent string describing the error, or an empty string if there was no error.

`srcText` of type `DOMString`, readonly
: The source of the line where the error occurred if available.

`url` of type `DOMString`, readonly
: The normalized URL of the document where the error occurred.

Appendix A: IDL Definitions

This appendix contains the complete OMG IDL [OMGIDL] for the Level 3 Document Object Model Content Model and Load and Save definitions.

The IDL files are also available as:
http://www.w3.org/TR/2001/WD-DOM-Level-3-CMLS-20010419/idl.zip

content-models.idl:

```
// File: content-models.idl

#ifndef _CONTENT-MODELS_IDL_
#define _CONTENT-MODELS_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module content-models
{

  typedef dom::DOMString DOMString;
  typedef dom::decimal decimal;
  typedef dom::int int;
  typedef dom::Node Node;
  typedef dom::nsElement nsElement;
  typedef dom::DOMImplementation DOMImplementation;
  typedef dom::Element,NodeCM Element,NodeCM;
  typedef dom::Text,NodeCM Text,NodeCM;
  typedef dom::DocumentType,NodeCM DocumentType,NodeCM;
  typedef dom::Attr,NodeCM Attr,NodeCM;

  interface CMModel;
  interface CMChildren;
  interface DOMErrorHandler;
  interface DOMLocator;

  interface CMNode {
    const unsigned short      CM_ELEMENT_DECLARATION         = 1;
    const unsigned short      CM_ATTRIBUTE_DECLARATION       = 2;
    const unsigned short      CM_NOTATION_DECLARATION        = 3;
    const unsigned short      CM_ENTITY_DECLARATION          = 4;
    const unsigned short      CM_CHILDREN                    = 5;
    const unsigned short      CM_MODEL                       = 6;
    const unsigned short      CM_EXTERNALMODEL               = 7;
    readonly attribute unsigned short   cmNodeType;
             attribute CMModel          ownerCMModel;
             attribute DOMString        nodeName;
             attribute DOMString        prefix;
             attribute DOMString        localName;
             attribute DOMString        namespaceURI;
    CMNode             clone();
  };
```

```
interface CMNodeList {
};

interface CMNamedNodeMap {
};

interface CMDataType {
  const short               STRING_DATATYPE                = 1;
  short              getCMPrimitiveType();
};

interface CMPrimitiveType : CMDataType {
  const short               BOOLEAN_DATATYPE               = 2;
  const short               FLOAT_DATATYPE                 = 3;
  const short               DOUBLE_DATATYPE                = 4;
  const short               DECIMAL_DATATYPE               = 5;
  const short               HEXBINARY_DATATYPE             = 6;
  const short               BASE64BINARY_DATATYPE          = 7;
  const short               ANYURI_DATATYPE                = 8;
  const short               QNAME_DATATYPE                 = 9;
  const short               DURATION_DATATYPE              = 10;
  const short               DATETIME_DATATYPE              = 11;
  const short               DATE_DATATYPE                  = 12;
  const short               TIME_DATATYPE                  = 13;
  const short               YEARMONTH_DATATYPE             = 14;
  const short               YEAR_DATATYPE                  = 15;
  const short               MONTHDAY_DATATYPE              = 16;
  const short               DAY_DATATYPE                   = 17;
  const short               MONTH_DATATYPE                 = 18;
  const short               NOTATION_DATATYPE              = 19;
           attribute decimal          lowValue;
           attribute decimal          highValue;
};

interface CMElementDeclaration : CMNode {
           attribute CMDataType       elementType;
  readonly attribute boolean          isPCDataOnly;
           attribute DOMString        tagName;
  int                getContentType();
  CMChildren         getCMChildren();
  CMNamedNodeMap     getCMAttributes();
  CMNamedNodeMap     getCMGrandChildren();
};

interface CMChildren : CMNode {
  const unsigned long       UNBOUNDED                      = MAX_LONG;
  const unsigned short      NONE                           = 0;
  const unsigned short      SEQUENCE                       = 1;
  const unsigned short      CHOICE                         = 2;
           attribute unsigned short   listOperator;
           attribute unsigned long    minOccurs;
           attribute unsigned long    maxOccurs;
           attribute CMNodeList       subModels;
  CMNode             removeCMNode(in unsigned long nodeIndex);
  int                insertCMNode(in unsigned long nodeIndex,
                                  in CMNode newNode);
  int                appendCMNode(in CMNode newNode);
```

```
};

interface CMAttributeDeclaration : CMNode {
  const short               NO_VALUE_CONSTRAINT            = 0;
  const short               DEFAULT_VALUE_CONSTRAINT       = 1;
  const short               FIXED_VALUE_CONSTRAINT         = 2;
           attribute DOMString        attrName;
           attribute CMDataType       attrType;
           attribute DOMString        attributeValue;
           attribute DOMString        enumAttr;
           attribute CMNodeList       ownerElement;
           attribute short            constraintType;
};

interface CMEntityDeclaration : CMNode {
  const short               INTERNAL_ENTITY                = 1;
  const short               EXTERNAL_ENTITY                = 2;
           attribute short            entityType;
           attribute DOMString        entityName;
           attribute DOMString        entityValue;
           attribute DOMString        systemId;
           attribute DOMString        publicId;
           attribute DOMString        notationName;
};

interface CMNotationDeclaration : CMNode {
           attribute DOMString        notationName;
           attribute DOMString        systemId;
           attribute DOMString        publicId;
};

interface Document {
  void               setErrorHandler(in DOMErrorHandler handler);
};

interface DocumentCM : Document {
  const short               WF_CHECK                       = 1;
  const short               NS_WF_CHECK                    = 2;
  const short               PARTIAL_VALIDITY_CHECK         = 3;
  const short               STRICT_VALIDITY_CHECK          = 4;
           attribute boolean          continuousValidityChecking;
           attribute short            wfValidityCheckLevel;
  int                numCMs();
  CMModel            getInternalCM();
  CMNodeList         getCMs();
  CMModel            getActiveCM();
  void               addCM(in CMModel cm);
  void               removeCM(in CMModel cm);
  boolean            activateCM(in CMModel cm);
};

interface DOMErrorHandler {
  void               warning(in DOMLocator where,
                             in DOMString how,
                             in DOMString why)
                                      raises(dom::DOMSystemException);
  void               fatalError(in DOMLocator where,
```

```
                                in DOMString how,
                                in DOMString why)
                                      raises(dom::DOMSystemException);
  void               error(in DOMLocator where,
                           in DOMString how,
                           in DOMString why)
                                      raises(dom::DOMSystemException);
};

interface DOMLocator {
  int                getColumnNumber();
  int                getLineNumber();
  DOMString          getPublicID();
  DOMString          getSystemID();
  Node               getNode();
};

interface CMModel : CMNode {
  readonly attribute boolean          isNamespaceAware;
           attribute CMElementDeclaration  rootElementDecl;
  DOMString          getLocation();
  nsElement          getCMNamespace();
  CMNamedNodeMap     getCMNodes();
  boolean            removeNode(in CMNode node);
  boolean            insertBefore(in CMNode newNode,
                                  in CMNode refNode);
  boolean            validate();
  CMElementDeclaration createCMElementDeclaration(inout DOMString namespaceURI,
                                                  in DOMString qualifiedElementName,
                                                  in int contentSpec)
                                      raises(dom::DOMException);
  CMAttributeDeclaration createCMAttributeDeclaration(inout DOMString namespaceURI,
                                                      in DOMString qualifiedName)
                                      raises(dom::DOMException);
  CMNotationDeclaration createCMNotationDeclaration(in DOMString name,
                                                    in DOMString systemIdentifier,
                                                    inout DOMString publicIdentifier)
                                      raises(dom::DOMException);
  CMEntityDeclaration createCMEntityDeclaration(in DOMString name)
                                      raises(dom::DOMException);
  CMChildren         createCMChildren(in unsigned long minOccurs,
                                      in unsigned long maxOccurs,
                                      inout unsigned short operator)
                                      raises(dom::DOMException);
};

interface CMExternalModel : CMModel {
};

interface DOMImplementationCM : DOMImplementation {
  CMModel            createCM();
  CMExternalModel    createExternalCM();
};

interface NodeCM : Node {
  boolean            canInsertBefore(in Node newChild,
                                     in Node refChild)
```

```
                                 raises(dom::DOMException);
  boolean            canRemoveChild(in Node oldChild)
                                 raises(dom::DOMException);
  boolean            canReplaceChild(in Node newChild,
                                in Node oldChild)
                                 raises(dom::DOMException);
  boolean            canAppendChild(in Node newChild)
                                 raises(dom::DOMException);
  boolean            isValid()
                                 raises(dom::DOMException);
};

interface ElementCM : Element,NodeCM {
  int                contentType();
  CMElementDeclaration getElementDeclaration()
                                 raises(dom::DOMException);
  boolean            canSetAttribute(in DOMString attrname,
                                in DOMString attrval);
  boolean            canSetAttributeNode(in Node node);
  boolean            canSetAttributeNodeNS(in Node node);
  boolean            canSetAttributeNS(in DOMString attrname,
                                  in DOMString attrval,
                                  in DOMString namespaceURI,
                                  in DOMString localName);
  boolean            canRemoveAttribute(in DOMString attrname);
  boolean            canRemoveAttributeNS(in DOMString attrname,
                                     inout DOMString namespaceURI);
  boolean            canRemoveAttributeNode(in Node node);
};

interface CharacterDataCM : Text,NodeCM {
  boolean            isWhitespaceOnly();
  boolean            canSetData(in unsigned long offset,
                           in DOMString arg)
                                 raises(dom::DOMException);
  boolean            canAppendData(in DOMString arg)
                                 raises(dom::DOMException);
  boolean            canReplaceData(in unsigned long offset,
                               in unsigned long count,
                               in DOMString arg)
                                 raises(dom::DOMException);
  boolean            canInsertData(in unsigned long offset,
                              in DOMString arg)
                                 raises(dom::DOMException);
  boolean            canDeleteData(in unsigned long offset,
                              in DOMString arg)
                                 raises(dom::DOMException);
};

interface DocumentTypeCM : DocumentType,NodeCM {
  boolean            isElementDefined(in DOMString elemTypeName);
  boolean            isElementDefinedNS(in DOMString elemTypeName,
                                   in DOMString namespaceURI,
                                   in DOMString localName);
  boolean            isAttributeDefined(in DOMString elemTypeName,
                                   in DOMString attrName);
  boolean            isAttributeDefinedNS(in DOMString elemTypeName,
```

```
                                            in DOMString attrName,
                                            in DOMString namespaceURI,
                                            in DOMString localName);
    boolean             isEntityDefined(in DOMString entName);
  };

  interface AttributeCM : Attr,NodeCM {
    CMAttributeDeclaration getAttributeDeclaration();
    CMNotationDeclaration getNotation()
                                        raises(dom::DOMException);
  };
};

#endif // _CONTENT-MODELS_IDL_
```

load-save.idl:

```
// File: load-save.idl

#ifndef _LOAD-SAVE_IDL_
#define _LOAD-SAVE_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module load-save
{

  typedef dom::DOMErrorHandler DOMErrorHandler;
  typedef dom::DOMString DOMString;
  typedef dom::Document Document;
  typedef dom::DOMInputStream DOMInputStream;
  typedef dom::DOMReader DOMReader;
  typedef dom::Element Element;
  typedef dom::DOMOutputStream DOMOutputStream;
  typedef dom::Node Node;

  interface DOMBuilder;
  interface DOMWriter;
  interface DOMEntityResolver;
  interface DOMBuilderFilter;
  interface DOMInputSource;

  interface DOMImplementationLS {
    DOMBuilder          createDOMBuilder();
    DOMWriter           createDOMWriter();
  };

  interface DOMBuilder {
           attribute DOMEntityResolver  entityResolver;
           attribute DOMErrorHandler  errorHandler;
           attribute DOMBuilderFilter  filter;
    void                setFeature(in DOMString name,
                                   in boolean state)
                                        raises(dom::DOMException);
    boolean             supportsFeature(in DOMString name);
```

```
  boolean            canSetFeature(in DOMString name,
                                   in boolean state);
  boolean            getFeature(in DOMString name)
                                      raises(dom::DOMException);
  Document           parseURI(in DOMString uri)
                                      raises(dom::DOMException,
                                             dom::DOMSystemException);
  Document           parseDOMInputSource(in DOMInputSource is)
                                      raises(dom::DOMException,
                                             dom::DOMSystemException);
};

interface DOMInputSource {
           attribute DOMInputStream   byteStream;
           attribute DOMReader        characterStream;
           attribute DOMString        encoding;
           attribute DOMString        publicId;
           attribute DOMString        systemId;
};

interface DOMEntityResolver {
  DOMInputSource     resolveEntity(in DOMString publicId,
                                   in DOMString systemId )
                                      raises(dom::DOMSystemException);
};

interface DOMBuilderFilter {
  boolean            startElement(in Element element);
  boolean            endElement(in Element element);
};

interface DOMWriter {
           attribute DOMString        encoding;
  readonly attribute DOMString        lastEncoding;
           attribute unsigned short   format;
  // Modified in DOM Level 3:
           attribute DOMString        newLine;
  void               writeNode(in DOMOutputStream destination,
                               in Node node)
                                      raises(dom::DOMSystemException);
};

interface DocumentLS {
           attribute boolean          async;
  void               abort();
  boolean            load(in DOMString url);
  boolean            loadXML(in DOMString source);
  DOMString          saveXML(in Node node)
                                      raises(dom::DOMException);
};

interface ParserErrorEvent {
  readonly attribute long             errorCode;
  readonly attribute long             filepos;
  readonly attribute long             line;
  readonly attribute long             linepos;
  readonly attribute DOMString        reason;
```

```
    readonly attribute DOMString         srcText;
    readonly attribute DOMString         url;
  };
};

#endif // _LOAD-SAVE_IDL_
```

Appendix B: Java Language Binding

This appendix contains the complete Java [Java] bindings for the Level 3 Document Object Model Content Model and Load and Save.

The Java files are also available as
http://www.w3.org/TR/2001/WD-DOM-Level-3-CMLS-20010419/java-binding.zip

org/w3c/dom/contentModel/CMModel.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.nsElement;
import org.w3c.dom.DOMException;

public interface CMModel extends CMNode {
    public boolean getIsNamespaceAware();

    public CMElementDeclaration getRootElementDecl();
    public void setRootElementDecl(CMElementDeclaration rootElementDecl);

    public String getLocation();

    public nsElement getCMNamespace();

    public CMNamedNodeMap getCMNodes();

    public boolean removeNode(CMNode node);

    public boolean insertBefore(CMNode newNode,
                                CMNode refNode);

    public boolean validate();

    public CMElementDeclaration createCMElementDeclaration(String namespaceURI,
                                                           String qualifiedElementName,
                                                           int contentSpec)
                                                           throws DOMException;

    public CMAttributeDeclaration createCMAttributeDeclaration(String namespaceURI,
                                                               String qualifiedName)
                                                               throws DOMException;

    public CMNotationDeclaration createCMNotationDeclaration(String name,
                                                             String systemIdentifier,
                                                             String publicIdentifier)
                                                             throws DOMException;

    public CMEntityDeclaration createCMEntityDeclaration(String name)
                                                         throws DOMException;

    public CMChildren createCMChildren(int minOccurs,
                                       int maxOccurs,
```

```
                                              short operator)
                                              throws DOMException;

}
```

org/w3c/dom/contentModel/CMExternalModel.java:

```
package org.w3c.dom.contentModel;

public interface CMExternalModel extends CMModel {
}
```

org/w3c/dom/contentModel/CMNode.java:

```
package org.w3c.dom.contentModel;

public interface CMNode {
    public static final short CM_ELEMENT_DECLARATION    = 1;
    public static final short CM_ATTRIBUTE_DECLARATION  = 2;
    public static final short CM_NOTATION_DECLARATION   = 3;
    public static final short CM_ENTITY_DECLARATION     = 4;
    public static final short CM_CHILDREN               = 5;
    public static final short CM_MODEL                  = 6;
    public static final short CM_EXTERNALMODEL          = 7;
    public short getCmNodeType();

    public CMModel getOwnerCMModel();
    public void setOwnerCMModel(CMModel ownerCMModel);

    public String getNodeName();
    public void setNodeName(String nodeName);

    public String getPrefix();
    public void setPrefix(String prefix);

    public String getLocalName();
    public void setLocalName(String localName);

    public String getNamespaceURI();
    public void setNamespaceURI(String namespaceURI);

    public CMNode clone();

}
```

org/w3c/dom/contentModel/CMNodeList.java:

```
package org.w3c.dom.contentModel;

public interface CMNodeList {
}
```

org/w3c/dom/contentModel/CMNamedNodeMap.java:

```
package org.w3c.dom.contentModel;

public interface CMNamedNodeMap {
}
```

org/w3c/dom/contentModel/CMDataType.java:

```
package org.w3c.dom.contentModel;

public interface CMDataType {
    public static final short STRING_DATATYPE           = 1;
    public short getCMPrimitiveType();

}
```

org/w3c/dom/contentModel/CMPrimitiveType.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.decimal;

public interface CMPrimitiveType extends CMDataType {
    public static final short BOOLEAN_DATATYPE          = 2;
    public static final short FLOAT_DATATYPE            = 3;
    public static final short DOUBLE_DATATYPE           = 4;
    public static final short DECIMAL_DATATYPE          = 5;
    public static final short HEXBINARY_DATATYPE        = 6;
    public static final short BASE64BINARY_DATATYPE     = 7;
    public static final short ANYURI_DATATYPE           = 8;
    public static final short QNAME_DATATYPE            = 9;
    public static final short DURATION_DATATYPE         = 10;
    public static final short DATETIME_DATATYPE         = 11;
    public static final short DATE_DATATYPE             = 12;
    public static final short TIME_DATATYPE             = 13;
    public static final short YEARMONTH_DATATYPE        = 14;
    public static final short YEAR_DATATYPE             = 15;
    public static final short MONTHDAY_DATATYPE         = 16;
    public static final short DAY_DATATYPE              = 17;
    public static final short MONTH_DATATYPE            = 18;
    public static final short NOTATION_DATATYPE         = 19;
    public decimal getLowValue();
    public void setLowValue(decimal lowValue);

    public decimal getHighValue();
    public void setHighValue(decimal highValue);

}
```

org/w3c/dom/contentModel/CMElementDeclaration.java:

```
package org.w3c.dom.contentModel;

public interface CMElementDeclaration extends CMNode {
    public CMDataType getElementType();
    public void setElementType(CMDataType elementType);

    public boolean getIsPCDataOnly();

    public String getTagName();
    public void setTagName(String tagName);

    public int getContentType();

    public CMChildren getCMChildren();

    public CMNamedNodeMap getCMAttributes();

    public CMNamedNodeMap getCMGrandChildren();

}
```

org/w3c/dom/contentModel/CMChildren.java:

```
package org.w3c.dom.contentModel;

public interface CMChildren extends CMNode {
    public static final int UNBOUNDED                = MAX_LONG;
    public static final short NONE                     = 0;
    public static final short SEQUENCE                 = 1;
    public static final short CHOICE                   = 2;
    public short getListOperator();
    public void setListOperator(short listOperator);

    public int getMinOccurs();
    public void setMinOccurs(int minOccurs);

    public int getMaxOccurs();
    public void setMaxOccurs(int maxOccurs);

    public CMNodeList getSubModels();
    public void setSubModels(CMNodeList subModels);

    public CMNode removeCMNode(int nodeIndex);

    public int insertCMNode(int nodeIndex,
                            CMNode newNode);

    public int appendCMNode(CMNode newNode);

}
```

org/w3c/dom/contentModel/CMAttributeDeclaration.java:

```
package org.w3c.dom.contentModel;

public interface CMAttributeDeclaration extends CMNode {
    public static final short NO_VALUE_CONSTRAINT       = 0;
    public static final short DEFAULT_VALUE_CONSTRAINT  = 1;
    public static final short FIXED_VALUE_CONSTRAINT    = 2;
    public String getAttrName();
    public void setAttrName(String attrName);

    public CMDataType getAttrType();
    public void setAttrType(CMDataType attrType);

    public String getAttributeValue();
    public void setAttributeValue(String attributeValue);

    public String getEnumAttr();
    public void setEnumAttr(String enumAttr);

    public CMNodeList getOwnerElement();
    public void setOwnerElement(CMNodeList ownerElement);

    public short getConstraintType();
    public void setConstraintType(short constraintType);

}
```

org/w3c/dom/contentModel/CMEntityDeclaration.java:

```
package org.w3c.dom.contentModel;

public interface CMEntityDeclaration extends CMNode {
    public static final short INTERNAL_ENTITY           = 1;
    public static final short EXTERNAL_ENTITY           = 2;
    public short getEntityType();
    public void setEntityType(short entityType);

    public String getEntityName();
    public void setEntityName(String entityName);

    public String getEntityValue();
    public void setEntityValue(String entityValue);

    public String getSystemId();
    public void setSystemId(String systemId);

    public String getPublicId();
    public void setPublicId(String publicId);

    public String getNotationName();
    public void setNotationName(String notationName);

}
```

org/w3c/dom/contentModel/CMNotationDeclaration.java:

```
package org.w3c.dom.contentModel;

public interface CMNotationDeclaration extends CMNode {
    public String getNotationName();
    public void setNotationName(String notationName);

    public String getSystemId();
    public void setSystemId(String systemId);

    public String getPublicId();
    public void setPublicId(String publicId);

}
```

org/w3c/dom/contentModel/Document.java:

```
package org.w3c.dom.contentModel;

public interface Document {
    public void setErrorHandler(DOMErrorHandler handler);

}
```

org/w3c/dom/contentModel/DocumentCM.java:

```
package org.w3c.dom.contentModel;

public interface DocumentCM extends Document {
    public static final short WF_CHECK                  = 1;
    public static final short NS_WF_CHECK               = 2;
    public static final short PARTIAL_VALIDITY_CHECK    = 3;
    public static final short STRICT_VALIDITY_CHECK     = 4;
    public boolean getContinuousValidityChecking();
    public void setContinuousValidityChecking(boolean continuousValidityChecking);

    public short getWfValidityCheckLevel();
    public void setWfValidityCheckLevel(short wfValidityCheckLevel);

    public int numCMs();

    public CMModel getInternalCM();

    public CMNodeList getCMs();

    public CMModel getActiveCM();

    public void addCM(CMModel cm);

    public void removeCM(CMModel cm);

    public boolean activateCM(CMModel cm);

}
```

org/w3c/dom/contentModel/DOMImplementationCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.DOMImplementation;

public interface DOMImplementationCM extends DOMImplementation {
    public CMModel createCM();

    public CMExternalModel createExternalCM();

}
```

org/w3c/dom/contentModel/NodeCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface NodeCM extends Node {
    public boolean canInsertBefore(Node newChild,
                                   Node refChild)
                                   throws DOMException;

    public boolean canRemoveChild(Node oldChild)
                                  throws DOMException;

    public boolean canReplaceChild(Node newChild,
                                   Node oldChild)
                                   throws DOMException;

    public boolean canAppendChild(Node newChild)
                                  throws DOMException;

    public boolean isValid()
                          throws DOMException;

}
```

org/w3c/dom/contentModel/ElementCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.Node;
import org.w3c.dom.Element,NodeCM;
import org.w3c.dom.DOMException;

public interface ElementCM extends Element,NodeCM {
    public int contentType();

    public CMElementDeclaration getElementDeclaration()
                                                   throws DOMException;

    public boolean canSetAttribute(String attrname,
```

```
                                   String attrval);

    public boolean canSetAttributeNode(Node node);

    public boolean canSetAttributeNodeNS(Node node);

    public boolean canSetAttributeNS(String attrname,
                                     String attrval,
                                     String namespaceURI,
                                     String localName);

    public boolean canRemoveAttribute(String attrname);

    public boolean canRemoveAttributeNS(String attrname,
                                        String namespaceURI);

    public boolean canRemoveAttributeNode(Node node);

}
```

org/w3c/dom/contentModel/CharacterDataCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.Text,NodeCM;
import org.w3c.dom.DOMException;

public interface CharacterDataCM extends Text,NodeCM {
    public boolean isWhitespaceOnly();

    public boolean canSetData(int offset,
                              String arg)
                              throws DOMException;

    public boolean canAppendData(String arg)
                                 throws DOMException;

    public boolean canReplaceData(int offset,
                                  int count,
                                  String arg)
                                  throws DOMException;

    public boolean canInsertData(int offset,
                                 String arg)
                                 throws DOMException;

    public boolean canDeleteData(int offset,
                                 String arg)
                                 throws DOMException;

}
```

org/w3c/dom/contentModel/DocumentTypeCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.DocumentType,NodeCM;

public interface DocumentTypeCM extends DocumentType,NodeCM {
    public boolean isElementDefined(String elemTypeName);

    public boolean isElementDefinedNS(String elemTypeName,
                                      String namespaceURI,
                                      String localName);

    public boolean isAttributeDefined(String elemTypeName,
                                      String attrName);

    public boolean isAttributeDefinedNS(String elemTypeName,
                                        String attrName,
                                        String namespaceURI,
                                        String localName);

    public boolean isEntityDefined(String entName);

}
```

org/w3c/dom/contentModel/AttributeCM.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.Attr,NodeCM;
import org.w3c.dom.DOMException;

public interface AttributeCM extends Attr,NodeCM {
    public CMAttributeDeclaration getAttributeDeclaration();

    public CMNotationDeclaration getNotation()
                                             throws DOMException;

}
```

org/w3c/dom/contentModel/DOMErrorHandler.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.DOMSystemException;

public interface DOMErrorHandler {
    public void warning(DOMLocator where,
                        String how,
                        String why)
                        throws DOMSystemException;

    public void fatalError(DOMLocator where,
                           String how,
                           String why)
```

```
                            throws DOMSystemException;

    public void error(DOMLocator where,
                      String how,
                      String why)
                      throws DOMSystemException;

}
```

org/w3c/dom/contentModel/DOMLocator.java:

```
package org.w3c.dom.contentModel;

import org.w3c.dom.Node;

public interface DOMLocator {
    public int getColumnNumber();

    public int getLineNumber();

    public String getPublicID();

    public String getSystemID();

    public Node getNode();

}
```

org/w3c/dom/loadSave/DOMImplementationLS.java:

```
package org.w3c.dom.loadSave;

public interface DOMImplementationLS {
    public DOMBuilder createDOMBuilder();

    public DOMWriter createDOMWriter();

}
```

org/w3c/dom/loadSave/DOMBuilder.java:

```
package org.w3c.dom.loadSave;

import org.w3c.dom.DOMErrorHandler;
import org.w3c.dom.Document;
import org.w3c.dom.DOMSystemException;
import org.w3c.dom.DOMException;

public interface DOMBuilder {
    public DOMEntityResolver getEntityResolver();
    public void setEntityResolver(DOMEntityResolver entityResolver);

    public DOMErrorHandler getErrorHandler();
    public void setErrorHandler(DOMErrorHandler errorHandler);
```

```
    public DOMBuilderFilter getFilter();
    public void setFilter(DOMBuilderFilter filter);

    public void setFeature(String name,
                           boolean state)
                           throws DOMException;

    public boolean supportsFeature(String name);

    public boolean canSetFeature(String name,
                                 boolean state);

    public boolean getFeature(String name)
                              throws DOMException;

    public Document parseURI(String uri)
                             throws DOMException, DOMSystemException;

    public Document parseDOMInputSource(DOMInputSource is)
                                        throws DOMException, DOMSystemException;

}
```

org/w3c/dom/loadSave/DOMInputSource.java:

```
package org.w3c.dom.loadSave;

public interface DOMInputSource {
    public java.io.InputStream getByteStream();
    public void setByteStream(java.io.InputStream byteStream);

    public java.io.Reader getCharacterStream();
    public void setCharacterStream(java.io.Reader characterStream);

    public String getEncoding();
    public void setEncoding(String encoding);

    public String getPublicId();
    public void setPublicId(String publicId);

    public String getSystemId();
    public void setSystemId(String systemId);

}
```

org/w3c/dom/loadSave/DOMEntityResolver.java:

```
package org.w3c.dom.loadSave;

import org.w3c.dom.DOMSystemException;

public interface DOMEntityResolver {
    public DOMInputSource resolveEntity(String publicId,
```

```
                                        String systemId )
                                        throws DOMSystemException;

}
```

org/w3c/dom/loadSave/DOMBuilderFilter.java:

```
package org.w3c.dom.loadSave;

import org.w3c.dom.Element;

public interface DOMBuilderFilter {
    public boolean startElement(Element element);

    public boolean endElement(Element element);

}
```

org/w3c/dom/loadSave/DOMWriter.java:

```
package org.w3c.dom.loadSave;

import org.w3c.dom.Node;
import org.w3c.dom.DOMSystemException;

public interface DOMWriter {
    public String getEncoding();
    public void setEncoding(String encoding);

    public String getLastEncoding();

    public short getFormat();
    public void setFormat(short format);

    public String getNewLine();
    public void setNewLine(String newLine);

    public void writeNode(java.io.OutputStream destination,
                          Node node)
                          throws DOMSystemException;

}
```

org/w3c/dom/loadSave/DocumentLS.java:

```
package org.w3c.dom.loadSave;

import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface DocumentLS {
    public boolean getAsync();
    public void setAsync(boolean async);

    public void abort();
```

```
    public boolean load(String url);

    public boolean loadXML(String source);

    public String saveXML(Node node)
                          throws DOMException;

}
```

org/w3c/dom/loadSave/ParserErrorEvent.java:

```
package org.w3c.dom.loadSave;

public interface ParserErrorEvent {
    public int getErrorCode();

    public int getFilepos();

    public int getLine();

    public int getLinepos();

    public String getReason();

    public String getSrcText();

    public String getUrl();

}
```

org/w3c/dom/loadSave/ParserErrorEvent.java:

Appendix C: ECMA Script Language Binding

This appendix contains the complete ECMA Script [ECMAScript] binding for the Level 3 Document Object Model Content Model and Load and Save definitions.

Object **CMModel**

CMModel has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.

The **CMModel** object has the following properties:

isNamespaceAware

This read-only property is of type **Boolean**.

rootElementDecl

This property is a **CMElementDeclaration** object.

The **CMModel** object has the following methods:

getLocation()

This method returns a **String**.

getCMNamespace()

This method returns a **nsElement** object.

getCMNodes()

This method returns a **CMNamedNodeMap** object.

removeNode(node)

This method returns a **Boolean**.
The **node** parameter is a **CMNode** object.

insertBefore(newNode, refNode)

This method returns a **Boolean**.
The **newNode** parameter is a **CMNode** object.
The **refNode** parameter is a **CMNode** object.

validate()

This method returns a **Boolean**.

createCMElementDeclaration(namespaceURI, qualifiedElementName, contentSpec)

This method returns a **CMElementDeclaration** object.
The **namespaceURI** parameter is of type **String**.
The **qualifiedElementName** parameter is of type **String**.
The **contentSpec** parameter is a **int** object.
This method can raise a **DOMException** object.

createCMAttributeDeclaration(namespaceURI, qualifiedName)

This method returns a **CMAttributeDeclaration** object.
The **namespaceURI** parameter is of type **String**.
The **qualifiedName** parameter is of type **String**.
This method can raise a **DOMException** object.

createCMNotationDeclaration(name, systemIdentifier, publicIdentifier)

This method returns a **CMNotationDeclaration** object.
The **name** parameter is of type **String**.
The **systemIdentifier** parameter is of type **String**.
The **publicIdentifier** parameter is of type **String**.
This method can raise a **DOMException** object.

createCMEntityDeclaration(name)
This method returns a **CMEntityDeclaration** object.
The **name** parameter is of type **String**.
This method can raise a **DOMException** object.
createCMChildren(minOccurs, maxOccurs, operator)
This method returns a **CMChildren** object.
The **minOccurs** parameter is of type **Number**.
The **maxOccurs** parameter is of type **Number**.
The **operator** parameter is of type **Number**.
This method can raise a **DOMException** object.

Object **CMExternalModel**
CMExternalModel has the all the properties and methods of the **CMModel** object as well as the properties and methods defined below.

Prototype Object **CMNode**
The **CMNode** class has the following constants:
CMNode.CM_ELEMENT_DECLARATION
This constant is of type **Number** and its value is **1**.
CMNode.CM_ATTRIBUTE_DECLARATION
This constant is of type **Number** and its value is **2**.
CMNode.CM_NOTATION_DECLARATION
This constant is of type **Number** and its value is **3**.
CMNode.CM_ENTITY_DECLARATION
This constant is of type **Number** and its value is **4**.
CMNode.CM_CHILDREN
This constant is of type **Number** and its value is **5**.
CMNode.CM_MODEL
This constant is of type **Number** and its value is **6**.
CMNode.CM_EXTERNALMODEL
This constant is of type **Number** and its value is **7**.

Object **CMNode**
The **CMNode** object has the following properties:
cmNodeType
This read-only property is of type **Number**.
ownerCMModel
This property is a **CMModel** object.
nodeName
This property is of type **String**.
prefix
This property is of type **String**.
localName
This property is of type **String**.
namespaceURI
This property is of type **String**.
The **CMNode** object has the following methods:
clone()
This method returns a **CMNode** object.

Object **CMNodeList**
Object **CMNamedNodeMap**
Prototype Object **CMDataType**
The **CMDataType** class has the following constants:
CMDataType.STRING_DATATYPE
This constant is of type **short** and its value is **1**.
Object **CMDataType**
The **CMDataType** object has the following methods:
getCMPrimitiveType()
This method returns a **short** object.
Prototype Object **CMPrimitiveType**
The **CMPrimitiveType** class has the following constants:
CMPrimitiveType.BOOLEAN_DATATYPE
This constant is of type **short** and its value is **2**.
CMPrimitiveType.FLOAT_DATATYPE
This constant is of type **short** and its value is **3**.
CMPrimitiveType.DOUBLE_DATATYPE
This constant is of type **short** and its value is **4**.
CMPrimitiveType.DECIMAL_DATATYPE
This constant is of type **short** and its value is **5**.
CMPrimitiveType.HEXBINARY_DATATYPE
This constant is of type **short** and its value is **6**.
CMPrimitiveType.BASE64BINARY_DATATYPE
This constant is of type **short** and its value is **7**.
CMPrimitiveType.ANYURI_DATATYPE
This constant is of type **short** and its value is **8**.
CMPrimitiveType.QNAME_DATATYPE
This constant is of type **short** and its value is **9**.
CMPrimitiveType.DURATION_DATATYPE
This constant is of type **short** and its value is **10**.
CMPrimitiveType.DATETIME_DATATYPE
This constant is of type **short** and its value is **11**.
CMPrimitiveType.DATE_DATATYPE
This constant is of type **short** and its value is **12**.
CMPrimitiveType.TIME_DATATYPE
This constant is of type **short** and its value is **13**.
CMPrimitiveType.YEARMONTH_DATATYPE
This constant is of type **short** and its value is **14**.
CMPrimitiveType.YEAR_DATATYPE
This constant is of type **short** and its value is **15**.
CMPrimitiveType.MONTHDAY_DATATYPE
This constant is of type **short** and its value is **16**.
CMPrimitiveType.DAY_DATATYPE
This constant is of type **short** and its value is **17**.
CMPrimitiveType.MONTH_DATATYPE
This constant is of type **short** and its value is **18**.

CMPrimitiveType.NOTATION_DATATYPE
This constant is of type **short** and its value is **19**.

Object **CMPrimitiveType**

CMPrimitiveType has the all the properties and methods of the **CMDataType** object as well as the properties and methods defined below.

The **CMPrimitiveType** object has the following properties:

lowValue
This property is a **decimal** object.

highValue
This property is a **decimal** object.

Object **CMElementDeclaration**

CMElementDeclaration has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.

The **CMElementDeclaration** object has the following properties:

elementType
This property is a **CMDataType** object.

isPCDataOnly
This read-only property is of type **Boolean**.

tagName
This property is of type **String**.

The **CMElementDeclaration** object has the following methods:

getContentType()
This method returns a **int** object.

getCMChildren()
This method returns a **CMChildren** object.

getCMAttributes()
This method returns a **CMNamedNodeMap** object.

getCMGrandChildren()
This method returns a **CMNamedNodeMap** object.

Prototype Object **CMChildren**

The **CMChildren** class has the following constants:

CMChildren.UNBOUNDED
This constant is of type **Number** and its value is **MAX_LONG**.

CMChildren.NONE
This constant is of type **Number** and its value is **0**.

CMChildren.SEQUENCE
This constant is of type **Number** and its value is **1**.

CMChildren.CHOICE
This constant is of type **Number** and its value is **2**.

Object **CMChildren**

CMChildren has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.

The **CMChildren** object has the following properties:

listOperator
This property is of type **Number**.

minOccurs
This property is of type **Number**.
maxOccurs
This property is of type **Number**.
subModels
This property is a **CMNodeList** object.
The **CMChildren** object has the following methods:
removeCMNode(nodeIndex)
This method returns a **CMNode** object.
The **nodeIndex** parameter is of type **Number**.
insertCMNode(nodeIndex, newNode)
This method returns a **int** object.
The **nodeIndex** parameter is of type **Number**.
The **newNode** parameter is a **CMNode** object.
appendCMNode(newNode)
This method returns a **int** object.
The **newNode** parameter is a **CMNode** object.
Prototype Object **CMAttributeDeclaration**
The **CMAttributeDeclaration** class has the following constants:
CMAttributeDeclaration.NO_VALUE_CONSTRAINT
This constant is of type **short** and its value is **0**.
CMAttributeDeclaration.DEFAULT_VALUE_CONSTRAINT
This constant is of type **short** and its value is **1**.
CMAttributeDeclaration.FIXED_VALUE_CONSTRAINT
This constant is of type **short** and its value is **2**.
Object **CMAttributeDeclaration**
CMAttributeDeclaration has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.
The **CMAttributeDeclaration** object has the following properties:
attrName
This property is of type **String**.
attrType
This property is a **CMDataType** object.
attributeValue
This property is of type **String**.
enumAttr
This property is of type **String**.
ownerElement
This property is a **CMNodeList** object.
constraintType
This property is a **short** object.
Prototype Object **CMEntityDeclaration**
The **CMEntityDeclaration** class has the following constants:
CMEntityDeclaration.INTERNAL_ENTITY
This constant is of type **short** and its value is **1**.

CMEntityDeclaration.EXTERNAL_ENTITY
This constant is of type **short** and its value is **2**.

Object **CMEntityDeclaration**

CMEntityDeclaration has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.

The **CMEntityDeclaration** object has the following properties:

entityType
This property is a **short** object.

entityName
This property is of type **String**.

entityValue
This property is of type **String**.

systemId
This property is of type **String**.

publicId
This property is of type **String**.

notationName
This property is of type **String**.

Object **CMNotationDeclaration**

CMNotationDeclaration has the all the properties and methods of the **CMNode** object as well as the properties and methods defined below.

The **CMNotationDeclaration** object has the following properties:

notationName
This property is of type **String**.

systemId
This property is of type **String**.

publicId
This property is of type **String**.

Object **Document**

The **Document** object has the following methods:

setErrorHandler(handler)
This method has no return value.
The **handler** parameter is a **DOMErrorHandler** object.

Prototype Object **DocumentCM**

The **DocumentCM** class has the following constants:

DocumentCM.WF_CHECK
This constant is of type **short** and its value is **1**.

DocumentCM.NS_WF_CHECK
This constant is of type **short** and its value is **2**.

DocumentCM.PARTIAL_VALIDITY_CHECK
This constant is of type **short** and its value is **3**.

DocumentCM.STRICT_VALIDITY_CHECK
This constant is of type **short** and its value is **4**.

Object **DocumentCM**

DocumentCM has the all the properties and methods of the **Document** object as well as the properties and methods defined below.

The **DocumentCM** object has the following properties:

continuousValidityChecking

This property is of type **Boolean**.

wfValidityCheckLevel

This property is a **short** object.

The **DocumentCM** object has the following methods:

numCMs()

This method returns a **int** object.

getInternalCM()

This method returns a **CMModel** object.

getCMs()

This method returns a **CMNodeList** object.

getActiveCM()

This method returns a **CMModel** object.

addCM(cm)

This method has no return value.

The **cm** parameter is a **CMModel** object.

removeCM(cm)

This method has no return value.

The **cm** parameter is a **CMModel** object.

activateCM(cm)

This method returns a **Boolean**.

The **cm** parameter is a **CMModel** object.

Object **DOMImplementationCM**

DOMImplementationCM has the all the properties and methods of the **DOMImplementation** object as well as the properties and methods defined below.

The **DOMImplementationCM** object has the following methods:

createCM()

This method returns a **CMModel** object.

createExternalCM()

This method returns a **CMExternalModel** object.

Object **NodeCM**

NodeCM has the all the properties and methods of the **Node** object as well as the properties and methods defined below.

The **NodeCM** object has the following methods:

canInsertBefore(newChild, refChild)

This method returns a **Boolean**.

The **newChild** parameter is a **Node** object.

The **refChild** parameter is a **Node** object.

This method can raise a **DOMException** object.

canRemoveChild(oldChild)

This method returns a **Boolean**.

The **oldChild** parameter is a **Node** object.

This method can raise a **DOMException** object.

canReplaceChild(newChild, oldChild)

This method returns a **Boolean**.

The **newChild** parameter is a **Node** object.
The **oldChild** parameter is a **Node** object.
This method can raise a **DOMException** object.

canAppendChild(newChild)
This method returns a **Boolean**.
The **newChild** parameter is a **Node** object.
This method can raise a **DOMException** object.

isValid()
This method returns a **Boolean**.
This method can raise a **DOMException** object.

Object **ElementCM**

ElementCM has the all the properties and methods of the **Element,NodeCM** object as well as the properties and methods defined below.

The **ElementCM** object has the following methods:

contentType()
This method returns a **int** object.

getElementDeclaration()
This method returns a **CMElementDeclaration** object.
This method can raise a **DOMException** object.

canSetAttribute(attrname, attrval)
This method returns a **Boolean**.
The **attrname** parameter is of type **String**.
The **attrval** parameter is of type **String**.

canSetAttributeNode(node)
This method returns a **Boolean**.
The **node** parameter is a **Node** object.

canSetAttributeNodeNS(node)
This method returns a **Boolean**.
The **node** parameter is a **Node** object.

canSetAttributeNS(attrname, attrval, namespaceURI, localName)
This method returns a **Boolean**.
The **attrname** parameter is of type **String**.
The **attrval** parameter is of type **String**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

canRemoveAttribute(attrname)
This method returns a **Boolean**.
The **attrname** parameter is of type **String**.

canRemoveAttributeNS(attrname, namespaceURI)
This method returns a **Boolean**.
The **attrname** parameter is of type **String**.
The **namespaceURI** parameter is of type **String**.

canRemoveAttributeNode(node)
This method returns a **Boolean**.
The **node** parameter is a **Node** object.

Object **CharacterDataCM**

CharacterDataCM has the all the properties and methods of the **Text,NodeCM** object as well as the properties and methods defined below.

The **CharacterDataCM** object has the following methods:

isWhitespaceOnly()

This method returns a **Boolean**.

canSetData(offset, arg)

This method returns a **Boolean**.
The **offset** parameter is of type **Number**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException** object.

canAppendData(arg)

This method returns a **Boolean**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException** object.

canReplaceData(offset, count, arg)

This method returns a **Boolean**.
The **offset** parameter is of type **Number**.
The **count** parameter is of type **Number**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException** object.

canInsertData(offset, arg)

This method returns a **Boolean**.
The **offset** parameter is of type **Number**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException** object.

canDeleteData(offset, arg)

This method returns a **Boolean**.
The **offset** parameter is of type **Number**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException** object.

Object **DocumentTypeCM**

DocumentTypeCM has the all the properties and methods of the **DocumentType,NodeCM** object as well as the properties and methods defined below.

The **DocumentTypeCM** object has the following methods:

isElementDefined(elemTypeName)

This method returns a **Boolean**.
The **elemTypeName** parameter is of type **String**.

isElementDefinedNS(elemTypeName, namespaceURI, localName)

This method returns a **Boolean**.
The **elemTypeName** parameter is of type **String**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

isAttributeDefined(elemTypeName, attrName)

This method returns a **Boolean**.
The **elemTypeName** parameter is of type **String**.

The **attrName** parameter is of type **String**.
isAttributeDefinedNS(elemTypeName, attrName, namespaceURI, localName)
This method returns a **Boolean**.
The **elemTypeName** parameter is of type **String**.
The **attrName** parameter is of type **String**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.
isEntityDefined(entName)
This method returns a **Boolean**.
The **entName** parameter is of type **String**.

Object **AttributeCM**
AttributeCM has the all the properties and methods of the **Attr,NodeCM** object as well as the properties and methods defined below.
The **AttributeCM** object has the following methods:
getAttributeDeclaration()
This method returns a **CMAttributeDeclaration** object.
getNotation()
This method returns a **CMNotationDeclaration** object.
This method can raise a **DOMException** object.

Object **DOMErrorHandler**
The **DOMErrorHandler** object has the following methods:
warning(where, how, why)
This method has no return value.
The **where** parameter is a **DOMLocator** object.
The **how** parameter is of type **String**.
The **why** parameter is of type **String**.
This method can raise a **DOMSystemException** object.
fatalError(where, how, why)
This method has no return value.
The **where** parameter is a **DOMLocator** object.
The **how** parameter is of type **String**.
The **why** parameter is of type **String**.
This method can raise a **DOMSystemException** object.
error(where, how, why)
This method has no return value.
The **where** parameter is a **DOMLocator** object.
The **how** parameter is of type **String**.
The **why** parameter is of type **String**.
This method can raise a **DOMSystemException** object.

Object **DOMLocator**
The **DOMLocator** object has the following methods:
getColumnNumber()
This method returns a **int** object.
getLineNumber()
This method returns a **int** object.

getPublicID()
This method returns a **String**.
getSystemID()
This method returns a **String**.
getNode()
This method returns a **Node** object.

Object **DOMImplementationLS**
The **DOMImplementationLS** object has the following methods:
createDOMBuilder()
This method returns a **DOMBuilder** object.
createDOMWriter()
This method returns a **DOMWriter** object.
Object **DOMBuilder**
The **DOMBuilder** object has the following properties:
entityResolver
This property is a **DOMEntityResolver** object.
errorHandler
This property is a **DOMErrorHandler** object.
filter
This property is a **DOMBuilderFilter** object.
The **DOMBuilder** object has the following methods:
setFeature(name, state)
This method has no return value.
The **name** parameter is of type **String**.
The **state** parameter is of type **Boolean**.
This method can raise a **DOMException** object.
supportsFeature(name)
This method returns a **Boolean**.
The **name** parameter is of type **String**.
canSetFeature(name, state)
This method returns a **Boolean**.
The **name** parameter is of type **String**.
The **state** parameter is of type **Boolean**.
getFeature(name)
This method returns a **Boolean**.
The **name** parameter is of type **String**.
This method can raise a **DOMException** object.
parseURI(uri)
This method returns a **Document** object.
The **uri** parameter is of type **String**.
This method can raise a **DOMException** object or a **DOMSystemException** object.
parseDOMInputSource(is)
This method returns a **Document** object.
The **is** parameter is a **DOMInputSource** object.
This method can raise a **DOMException** object or a **DOMSystemException** object.

Object **DOMInputSource**
The **DOMInputSource** object has the following properties:
byteStream
This property is a **DOMInputStream** object.
characterStream
This property is a **DOMReader** object.
encoding
This property is of type **String**.
publicId
This property is of type **String**.
systemId
This property is of type **String**.
Object **DOMEntityResolver**
The **DOMEntityResolver** object has the following methods:
resolveEntity(publicId, systemId)
This method returns a **DOMInputSource** object.
The **publicId** parameter is of type **String**.
The **systemId** parameter is of type **String**.
This method can raise a **DOMSystemException** object.
Object **DOMBuilderFilter**
The **DOMBuilderFilter** object has the following methods:
startElement(element)
This method returns a **Boolean**.
The **element** parameter is a **Element** object.
endElement(element)
This method returns a **Boolean**.
The **element** parameter is a **Element** object.
Object **DOMWriter**
The **DOMWriter** object has the following properties:
encoding
This property is of type **String**.
lastEncoding
This read-only property is of type **String**.
format
This property is of type **Number**.
newLine
This property is of type **String**.
The **DOMWriter** object has the following methods:
writeNode(destination, node)
This method has no return value.
The **destination** parameter is a **DOMOutputStream** object.
The **node** parameter is a **Node** object.
This method can raise a **DOMSystemException** object.
Object **DocumentLS**
The **DocumentLS** object has the following properties:

async
This property is of type **Boolean**.
The **DocumentLS** object has the following methods:
abort()
This method has no return value.
load(url)
This method returns a **Boolean**.
The **url** parameter is of type **String**.
loadXML(source)
This method returns a **Boolean**.
The **source** parameter is of type **String**.
saveXML(node)
This method returns a **String**.
The **node** parameter is a **Node** object.
This method can raise a **DOMException** object.
Object **ParserErrorEvent**
The **ParserErrorEvent** object has the following properties:
errorCode
This read-only property is a **long** object.
filepos
This read-only property is a **long** object.
line
This read-only property is a **long** object.
linepos
This read-only property is a **long** object.
reason
This read-only property is of type **String**.
srcText
This read-only property is of type **String**.
url
This read-only property is of type **String**.

References

For the latest version of any W3C specification please consult the list of W3C Technical Reports available at http://www.w3.org/TR.

D.1: Normative references

ECMAScript
: ECMA (European Computer Manufacturers Association) ECMAScript Language Specification. Available at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

Java
: Sun Microsystems Inc. The Java Language Specification, James Gosling, Bill Joy, and Guy Steele, September 1996. Available at http://java.sun.com/docs/books/jls

OMGIDL
: OMG (Object Management Group) IDL (Interface Definition Language) defined in The Common Object Request Broker: Architecture and Specification, version 2.3.1, October 1999. Available from http://www.omg.org

D.1: Normative references

Index