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

Version 1.0

W3C Working Draft *01 September, 2000*

This version:
http://www.w3.org/TR/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20000901
(PostScript file, PDF file, plain text, ZIP file)
Latest version:
http://www.w3.org/TR/DOM-Level-3-Content-Models-and-Load-Save

Editors:
Ben Chang, *Oracle*
Andy Heninger, *IBM*
Joe Kesselman, *IBM*

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 document is a preliminary version of the Level 3 API.

It is a W3C Working Draft for review by W3C members and other interested parties and acts as a starting point for the future DOM Working Group, should it be approved or not by the W3C Members. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress".

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. Different modules of the Document Object Model have different editors.

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

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 recommend you provide URIs to the location from which the code is derived.)

1. Content Models and Validation

Editors

Ben Chang, Oracle
Joe Kesselman, IBM

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. The feature string for all the interfaces listed in this section is "CM".

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. Presently, it is still unclear whether XML Schemas will be ready in time to be supported in DOM Level 3, but 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. However, some kinds of information on which the DOM already relies, e.g., default values for attributes, will finally be given a visible representation here.

1.1.2. Use Cases and Requirements

The content model referenced in these use cases/requirements is an abstraction and does not refer to DTDs or XML Schemas or any transformations between the two.

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. Modify an existing content model because it needs to be updated.
2. CU2. Associating a content model (external and/or internal) with a document, or changing the current association.
3. CU3. Using the same external content model with several documents, without having to reload it.
4. CU4. Create a new content model.

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. Create a new content model object.

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, 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.
3. CMU3. Partial content model checking. For example, only certain portions of the content model need be validated.

Requirements:

1. CMR1. View and modify all parts of the content model.
2. CMR2. Validate the content model itself. For example, if an element/attribute is inserted incorrectly into the content model.
3. CMR3. Serialize the content model.
4. CMR4. Clone all or parts of an existing content model.
5. CMR5. 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 assistance necessary so that valid documents can be modified and remain valid.
2. DU2. For editing documents with an associated content model, provide the assistance necessary to transform an invalid document into a valid one.

Requirements:

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

General Issues:

1. I1. Namespace issues associated with the content model. To address namespaces, a `isNamespaceAware` attribute to the generic CM object has been added to help applications determine if qualified names are important. Note that this should not be interpreted as helping identify what the underlying content model is. A MathML example to show how namespaced documents will be validated will be added later.
2. I2. Multiple CMs being associated with a XML document. For validation, this could: 1) result in an exception; 2) a merged content model for the document to be validated against; 3) each content model for the document to be validated against separately. In this chapter, we have gone for the third choice, allowing the user to specify which content model to be active and allowing them to keep adding content models to a list associated with the document.
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 both CM and Load/Save. Agreement has been to utilize user-registered callbacks but other details to be worked out.

1.1.2.1. Content Model Interfaces

A list of the proposed Content Model data structures and functions follow, starting off with the data structures.

Interface *CMObject*

`CMObject` 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 a link to the external subset, i.e., `CMExternalObject` [p.12] , which may be a non-negative number linked together. It is possible, however, that none of these `CMExternalObjects` are active. An attribute available will be `isNamespaceAware` to determine if qualified names are important.

IDL Definition

```
interface CMObject {
};
```

Interface *CMExternalObject*

`CMExternalObject` 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. Opaque.

IDL Definition

```
interface CMExternalObject {
};
```

Interface *CMNode*

`CMNode`, or a CMObject Node, is analogous to a node in the parse tree, e.g., an element declaration. This can exist for both `CMExternalObject` [p.12] (include/ignore must be handled here) and `CMObject` [p.21] . It should handle the following:

```
interface CommentsPIsDeclaration { attribute ProcessingInstruction
pis; attribute Comment comments; }; interface Conditional
Declaration { attribute boolean includeIgnore; };
```

Opaque.

IDL Definition

```
interface CMNode {
};
```

Interface *CMNodeList*

`CMNodeList` is the CM analogue to `NodeList`; ordering is important, as opposed to `NamedCMNodeMap` [p.13] . Opaque.

IDL Definition

```
interface CMNodeList {
};
```

Interface *NamedCMNodeMap*

`NamedCMNodeMap` is the CM analogue to `NamedNodeMap`. Ordering is not important. Opaque.

IDL Definition

```
interface NamedCMNodeMap {
};
```

Interface *CMDataType*

`CMDataType` is a string for now, as in "int" or "float", hence no typechecking.

IDL Definition

```
interface CMDataType {
};
```

Interface *CMType*

`CMType` is a `CMNode` [p.22] 's node type. For example, one type could be `ElementDeclaration` [p.23] , composed of a tagname, content-type, etc. Others could be `ElementCMModel` [p.26] and `AttributeDeclaration` [p.14] .

IDL Definition

```
interface CMType {
};
```

Interface *ElementDeclaration*

The element name along with a description: empty, any, mixed, elements, PCDATA, in the context of a `CMNode` [p.22] .

IDL Definition

```
interface ElementDeclaration {
  readonly attribute DOMString        elementName;
           attribute DOMString        contentType;
           attribute NamedCMNodeMap   attributes;
};
```

Attributes

`attributes` of type `NamedCMNodeMap` [p.13]
: CM's analogy to `NamedNodeMap`.

`contentType` of type `DOMString`
: Empty, any, mixed, elements, PCDATA.

`elementName` of type `DOMString`, readonly
Name of element.

Interface *ElementCMModel*

An element in the context of a `CMNode` [p.22] .

IDL Definition

```
interface ElementCMModel {
          attribute DOMString        listOperator;
          attribute int              multiplicity;
          attribute int              lowValue;
          attribute int              highValue;
          attribute NamedCMNodeMap   subModels;
          attribute CMNodeList       definingElement;
};
```

Attributes

`definingElement` of type `CMNodeList` [p.12]
Which CMNode in the list defined the element.

`highValue` of type `int`
The high value in the value range.

`listOperator` of type `DOMString`
Operator list.

`lowValue` of type `int`
The low value in the value range.

`multiplicity` of type `int`
0 or 1 or many.

`subModels` of type `NamedCMNodeMap` [p.13]
Additional `CMNode` [p.22] s in which the element can be defined.

Interface *AttributeDeclaration*

An attribute in the context of a `CMNode` [p.22] .

IDL Definition

```
interface AttributeDeclaration {
  readonly attribute DOMString        attrName;
           attribute CMDataType       attrType;
           attribute DOMString        defaultValue;
           attribute DOMString        enumAttr;
           attribute CMNodeList       ownerElement;
};
```

Attributes

`attrName` of type `DOMString`, readonly
: Name of attribute.

`attrType` of type `CMDataType` [p.13]
: Datatype of the attribute.

`defaultValue` of type `DOMString`
: Default value, which can also be expressed as a range, with high and low values.

`enumAttr` of type `DOMString`
: Enumeration of attribute.

`ownerElement` of type `CMNodeList` [p.12]
: Owner element CMNode of attribute.

Interface *EntityDeclaration*

As in current DOM.

IDL Definition

```
interface EntityDeclaration {
};
```

This section contains "Validation and Other" methods common to both the document-editing and CM-editing worlds (includes `Document`, `DOMImplementation`, and `ErrorHandler` [p.19] methods).

Interface *DocumentCM*

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

IDL Definition

```
interface DocumentCM : Document {
  boolean           isValid();
  int               numCMs();
  CMObject          getInternalCM();
  CMExternalObject * getCMs();
  CMObject          getActiveCM();
  void              addCM(in CMObject cm);
  void              removeCM(in CMObject cm);
  boolean           activateCM(in CMObject cm);
};
```

Methods

`activateCM`
: Make the given `CMObject` [p.21] active. Note that if an user wants to activate one CM to get default attribute values and then activate another to do validation, an user can do that; however, only one CM is active at a time.

Parameters

`cm` of type `CMObject` [p.21]
CM to be active for the document. The `CMObject` points to a list of `CMExternalObject` [p.12] s; with this call, only the specified CM will be active.

Return Value

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

No Exceptions

`addCM`

Associate a `CMObject` [p.21] with a document. Can be invoked multiple times to result in a list of `CMExternalObject` [p.12] s. Note that only one sole internal `CMObject` is associated with the document, however, and that only one of the possible list of `CMExternalObjects` is active at any one time.

Parameters

`cm` of type `CMObject` [p.21]
CM to be associated with the document.

No return.

No Return Value

No Exceptions

`getActiveCM`

Find the active `CMExternalObject` [p.12] for a document.

Return Value

`CMObject` [p.21]	`CMObject` with a pointer to the active `CMExternalObject` [p.12] of document.

No Parameters

No Exceptions

`getCMs`

Obtains a list of `CMExternalObject` [p.12] s associated with a document from the `CMObject` [p.21] . This list arises when `addCM()` is invoked.

Return Value

`CMExternalObject *`	A list of `CMExternalObject` [p.12] s associated with a document.

No Parameters

No Exceptions

`getInternalCM`

Find the sole `CMObject` [p.21] of a document. Only one `CMObject` may be associated with the document.

Return Value

`CMObject` [p.21]	`CMObject`.

No Parameters

No Exceptions

`isValid`

Determines if XML document is valid.

Return Value

`boolean`	Valid or not.

No Parameters

No Exceptions

`numCMs`

Determines number of `CMExternalObject` [p.12] s associated with the document. Only one `CMObject` [p.21] can be associated with the document, but it may point to a list of `CMExternalObjects`.

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 `CMExternalObject` [p.12] . Can be invoked multiple times to remove a number of these in the list of `CMExternalObjects`.

Parameters

`cm` of type `CMObject` [p.21]

CM to be removed.

No return.

No Return Value

No Exceptions

Interface *DomImplementationCM*

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

IDL Definition

```
interface DomImplementationCM : DomImplementation {
  boolean             validate();
  CMObject            createCM();
  CMExternalObject    createExternalCM();
  CMObject            cloneCM(in CMObject cm);
  CMExternalObject    cloneExternalCM(in CMExternalObject cm);
};
```

Methods

`cloneCM`

Copies a `CMObject` [p.21] to another `CMObject`. The `CMObject` returned wouldn't be associated with a document.

Parameters

`cm` of type `CMObject` [p.21]

`CMObject` to be cloned.

Return Value

`CMObject` [p.21]	Cloned `CMObject` or NULL if failure.

No Exceptions

`cloneExternalCM`

Copies a `CMExternalObject` [p.12] to another `CMExternalObject`. The `CMExternalObject` returned wouldn't be associated with a document.

Parameters

`cm` of type `CMExternalObject` [p.12]

`CMObject` [p.21] to be cloned.

Return Value

`CMExternalObject` [p.12]	Cloned `CMObject` [p.21] or NULL if failure.

No Exceptions

`createCM`

Creates a CMObject.

Return Value

`CMObject` [p.21]	A NULL return indicates failure.

No Parameters

No Exceptions

`createExternalCM`

Creates a CMExternalObject.

Return Value

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

No Parameters

No Exceptions

`validate`

Determines if a CMObject or CMExternalObject itself is valid; note that within a CMObject, a pointer to a CMExternalObject can exist.

Return Value

`boolean`	Is the CM valid?

No Parameters

No Exceptions

Interface *ErrorHandler*

Basic interface for CM or Load/Save error handlers. If an application needs to implement customized error handling for 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 ErrorHandler {
  void               warning(in where DOMString,
                             in how DOMString,
                             in why DOMString)
                                        raises(DOMException2);
  void               fatalError(in where DOMString,
                                in how DOMString,
                                in why DOMString)
                                        raises(DOMException2);
  void               error(in where DOMString,
                           in how DOMString,
                           in why DOMString)
                                        raises(DOMException2);
};
```

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 do nothing. The application may use this method to report conditions that are not fatal errors, and processing may continue even after invoking this method.

Parameters

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

`DOMString` of type `how`
: How the fatal error occurred.

`DOMString` of type `why`
: Why the fatal error occurred.

Exceptions

`DOMException2`	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 DOMException2. The application must stop all further processing when this method has been invoked.

Parameters

`DOMString` of type `where`
: 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 base, public ID and system ID for the error location could be some of the information.

`DOMString` of type `how`
: How the fatal error occurred.

`DOMString` of type `why`
: Why the fatal error occurred.

Exceptions

`DOMException2`	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 do nothing. The application may use this method to report conditions that are not errors or fatal errors, and processing may continue even after invoking this method.

Parameters

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

`DOMString` of type `how`
: How the warning occurred.

`DOMString` of type `why`
: Why the warning occurred.

Exceptions

`DOMException2`	A subclass of DOMException.

No Return Value

This section contains "CM-editing" methods (includes `CMObject` [p.21] , `CMNode` [p.22] , `ElementDeclaration` [p.23] , and `ElementCMModel` [p.26] methods).

Interface *CMObject*

`CMObject` 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 a link to the external subset, i.e., `CMExternalObject` [p.12] , which may be a non-negative number linked together. It is possible, however, that none of these `CMExternalObjects` are active. An attribute available will be `isNamespaceAware` to determine if qualified names are important.

IDL Definition

```
interface CMObject {
  readonly attribute boolean          isNamespaceAware;
  nsElement           getCMNamespace();
  namedCMNodeMap      getCMElements();
  boolean             removeCMNode(in CMNode node);
  boolean             insertbeforeCMNode(in CMNode newnode,
                                         in CMNode parentnode);
};
```

Attributes

`isNamespaceAware` of type `boolean`, readonly
: To determine if qualified names are important.

Methods

`getCMElements`
: Retrieves `CMNode` [p.22] list of which all `CMNodes` of type element declaration.

Return Value

`namedCMNodeMap`	List of all `CMNodes` [p.22] of type element declaration.

No Parameters
No Exceptions

`getCMNamespace`
Determines namespace of `CMObject`.
Return Value

`nsElement`	Namespace of `CMObject`.

No Parameters
No Exceptions

`insertbeforeCMNode`
Insert `CMNode` [p.22] .
Parameters
`newnode` of type `CMNode` [p.22]
`CMNode` to be inserted.

`parentnode` of type `CMNode`
`CMNode` to be inserted before.

Return Value

`boolean`	`Success or failure..`

No Exceptions

`removeCMNode`
Removes `CMNode` [p.22] and its children, if any.
Parameters
`node` of type `CMNode` [p.22]
`CMNode` to be removed.

Return Value

`boolean`	`Success or failure..`

No Exceptions

Interface *CMNode*

CMNode, or a CMObject Node, is analogous to a node in the parse tree, e.g., an element declaration. This can exist for both CMExternalObject [p.12] (include/ignore must be handled here) and CMObject [p.21] . It should handle the following:

IDL Definition

```
interface CMNode {
  CMType              getCMNodeType();
};
```

Methods

getCMNodeType

Determines type of CMNode.

Return Value

CMType [p.13]	CMType of CMNode.

No Parameters

No Exceptions

Interface *ElementDeclaration*

The element name along with a description: empty, any, mixed, elements, PCDATA, in the context of a CMNode [p.22] .

IDL Definition

```
interface ElementDeclaration {
  int                 getContentType();
  ElementCMModel      getCMElement();
  namedCMNodeMap      getCMAttributes();
  namedCMNodeMap      getCMElementsChildren();
};
```

Methods

getCMAttributes

Gets list of all attributes for this CMNode [p.22] .

Return Value

namedCMNodeMap	Attributes list for this CMNode [p.22] .

No Parameters

No Exceptions

getCMElement

Gets content model of element.

Return Value

ElementCMModel [p.26]	Content model of element.

No Parameters
No Exceptions

`getCMElementsChildren`
Gets list of children of `CMNode` [p.22] .
Return Value

`namedCMNodeMap`	Children list for this `CMNode` [p.22] .

No Parameters
No Exceptions

`getContentType`
Gets content type, e.g., empty, any, mixed, elements, PCDATA, of an element within a `CMNode` [p.22] .
Return Value

`int`	Content type constant.

No Parameters
No Exceptions

Interface *ElementCMModel*

An element in the context of a `CMNode` [p.22] .
IDL Definition

```
interface ElementCMModel {
  ElementCMModel     setCMElementCardinality(in CMNode node,
                                             in int high,
                                             in int low);
  ElementCMModel     getCMElementCardinality(in CMNode node,
                                             out int high,
                                             out int low);
};
```

Methods
`getCMElementCardinality`
Gets cardinality range of element's value.
Parameters
`node` of type `CMNode` [p.22]
`CMNode` for values to be retrieved.

`high` of type `int`
High value to be retrieved.

`low` of type `int`
Low value to be retrieved.

Return Value

`ElementCMModel` [p.26]	Element in the context of a `CMNode` with its high and low values retrieved.

No Exceptions

`setCMElementCardinality`
Sets cardinality range of element's value.
Parameters
`node` of type `CMNode` [p.22]
`CMNode` for values to be inserted.

`high` of type `int`
High value to be inserted.

`low` of type `int`
Low value to be inserted.

Return Value

`ElementCMModel` [p.26]	Element in the context of a `CMNode` with its high and low values set.

No Exceptions

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

Interface *NodeCM*

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

```
interface NodeCM : Node {
  boolean            canInsertBefore();
  boolean            canRemoveChild();
  boolean            canReplaceChild();
  boolean            canAppendChild();
};
```

Methods
`canAppendChild`
Has the same args as `AppendChild`.
Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

`canInsertBefore`
Has the same args as `InsertBefore`.
Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

`canRemoveChild`
Has the same args as `RemoveChild`.
Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

`canReplaceChild`
Has the same args as `ReplaceChild`.
Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

Interface *ElementCMModel*

An element in the context of a `CMNode` [p.22] .
IDL Definition

```
interface ElementCMModel {
  boolean             isValid();
  int                 contentType();
  boolean             canSetAttribute(in DOMString attrname,
                                      in DOMString attrval);
  boolean             canSetAttributeNode();
};
```

Methods

`canSetAttribute`

Sets value for specified attribute.

Parameters

`attrname` of type `DOMString`

Name of attribute.

`attrval` of type `DOMString`

Value to be assigned to the attribute.

Return Value

`boolean`	Success or failure.

No Exceptions

`canSetAttributeNode`

Determines if attribute can be set.

Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

`contentType`

Determines element content type.

Return Value

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

No Parameters
No Exceptions

`isValid`

Determines if element is valid.

Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

Interface *TextCM*

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

IDL Definition

```
interface TextCM : Text {
  boolean            isWhitespaceOnly();
  boolean            canSetData();
  boolean            canAppendData();
  boolean            canReplaceData();
  boolean            canInsertData();
};
```

Methods

`canAppendData`

Determines if data can be appended.

Return Value

`boolean`	Success or failure.

No Parameters

No Exceptions

`canInsertData`

Determines if data can be inserted.

Return Value

`boolean`	Success or failure.

No Parameters

No Exceptions

`canReplaceData`

Determines if data can be replaced.

Return Value

`boolean`	Success or failure.

No Parameters

No Exceptions

`canSetData`

Determines if data can be set.

Return Value

`boolean`	Success or failure.

No Parameters
No Exceptions

`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 *DocumentCM*

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

IDL Definition

```
interface DocumentCM : Document {
  boolean            isElementDefined(in DOMString elemTypeName);
  boolean            isAttributeDefined(in DOMString elemTypeName,
                                        in DOMString attrName);
  boolean            isEntityDefined(in DOMString entName);
};
```

Methods

`isAttributeDefined`

Determines if an attribute is defined in the document.

Parameters

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

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

Return Value

`boolean`	Success or failure.

No Exceptions

`isElementDefined`

Determines if an element is defined in the document.

Parameters

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

Return Value

`boolean`	Success or failure.

No Exceptions

`isEntityDefined`

Determines if an entity is defined in the document.

Parameters

`entName` of type `DOMString`

Name of entity.

Return Value

`boolean`	Success or failure.

No Exceptions

1.1.2.2. 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.

1.1.2.3. 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 NSURI/localname but different prefix... or same nodename but different NSURI.

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 preceeds this point, those which would be valid given both preceeding 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.

1.1.2.4. 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 revalidate 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.

1.1.2.5. 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 next section) and "lightweight" WF checking, as requested by the caller. 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 consistant. That isn't required during normal DOM operation, since we perform "early binding" and thereafter refer to nodes primarily via their NSURI 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 occuring 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 possily provide expose some of the primitive (eg, string-checking) functions directly.

1.1.2. Use Cases and Requirements

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 internal subset), DOM Level 2 implementations do not process the Content Model's content. For DOM Level 3, the Content Model's content must be 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 load ed Content Model rather than reparsing 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.

Appendix A: IDL Definitions

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

The IDL files are also available as:
http://www.w3.org/TR/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20000901/idl.zip

contentModel.idl:

```
// File: contentModel.idl

#ifndef _CONTENTMODEL_IDL_
#define _CONTENTMODEL_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module contentModel
{

  typedef dom::DOMString DOMString;
  typedef dom::int int;
  typedef dom::where where;
  typedef dom::how how;
  typedef dom::why why;
  typedef dom::nsElement nsElement;
  typedef dom::namedCMNodeMap namedCMNodeMap;
  typedef dom::Document Document;
  typedef dom::CMExternalObject * CMExternalObject *;
  typedef dom::DomImplementation DomImplementation;
  typedef dom::Node Node;
  typedef dom::Text Text;

  interface CMObject {
  };

  interface CMExternalObject {
  };

  interface CMNode {
  };

  interface CMNodeList {
  };

  interface NamedCMNodeMap {
  };

  interface CMDataType {
  };

  interface CMType {
  };
```

contentModel.idl:

```
interface ElementDeclaration {
  readonly attribute DOMString        elementName;
           attribute DOMString        contentType;
           attribute NamedCMNodeMap   attributes;
};

interface ElementCMModel {
           attribute DOMString        listOperator;
           attribute int              multiplicity;
           attribute int              lowValue;
           attribute int              highValue;
           attribute NamedCMNodeMap   subModels;
           attribute CMNodeList       definingElement;
};

interface AttributeDeclaration {
  readonly attribute DOMString        attrName;
           attribute CMDataType       attrType;
           attribute DOMString        defaultValue;
           attribute DOMString        enumAttr;
           attribute CMNodeList       ownerElement;
};

interface EntityDeclaration {
};

interface ErrorHandler {
  void               warning(in where DOMString,
                             in how DOMString,
                             in why DOMString)
                                      raises(dom::DOMException2);
  void               fatalError(in where DOMString,
                                in how DOMString,
                                in why DOMString)
                                      raises(dom::DOMException2);
  void               error(in where DOMString,
                           in how DOMString,
                           in why DOMString)
                                      raises(dom::DOMException2);
};

interface CMObject {
  readonly attribute boolean          isNamespaceAware;
  nsElement          getCMNamespace();
  namedCMNodeMap     getCMElements();
  boolean            removeCMNode(in CMNode node);
  boolean            insertbeforeCMNode(in CMNode newnode,
                                        in CMNode parentnode);
};

interface CMNode {
  CMType             getCMNodeType();
};

interface ElementDeclaration {
  int                getContentType();
```

```
  ElementCMModel     getCMElement();
  namedCMNodeMap     getCMAttributes();
  namedCMNodeMap     getCMElementsChildren();
};

interface ElementCMModel {
  ElementCMModel     setCMElementCardinality(in CMNode node,
                                             in int high,
                                             in int low);
  ElementCMModel     getCMElementCardinality(in CMNode node,
                                             out int high,
                                             out int low);
};

interface ElementCMModel {
  boolean            isValid();
  int                contentType();
  boolean            canSetAttribute(in DOMString attrname,
                                     in DOMString attrval);
  boolean            canSetAttributeNode();
};

interface DocumentCM : Document {
  boolean            isValid();
  int                numCMs();
  CMObject           getInternalCM();
  CMExternalObject * getCMs();
  CMObject           getActiveCM();
  void               addCM(in CMObject cm);
  void               removeCM(in CMObject cm);
  boolean            activateCM(in CMObject cm);
};

interface DomImplementationCM : DomImplementation {
  boolean            validate();
  CMObject           createCM();
  CMExternalObject   createExternalCM();
  CMObject           cloneCM(in CMObject cm);
  CMExternalObject   cloneExternalCM(in CMExternalObject cm);
};

interface NodeCM : Node {
  boolean            canInsertBefore();
  boolean            canRemoveChild();
  boolean            canReplaceChild();
  boolean            canAppendChild();
};

interface TextCM : Text {
  boolean            isWhitespaceOnly();
  boolean            canSetData();
  boolean            canAppendData();
  boolean            canReplaceData();
  boolean            canInsertData();
};

interface DocumentCM : Document {
```

```
    boolean             isElementDefined(in DOMString elemTypeName);
    boolean             isAttributeDefined(in DOMString elemTypeName,
                                          in DOMString attrName);
    boolean             isEntityDefined(in DOMString entName);
  };
};

#endif // _CONTENTMODEL_IDL_
```

Appendix B: Java Language Binding

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

The Java files are also available as
http://www.w3.org/TR/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20000830/java-binding.zip

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

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

public interface CMObject {
}
```

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

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

public interface CMExternalObject {
}
```

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

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

public interface CMNode {
}
```

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

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

public interface CMNodeList {
}
```

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

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

public interface NamedCMNodeMap {
}
```

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

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

public interface CMDataType {
}
```

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

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

public interface CMType {
}
```

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

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

public interface ElementDeclaration {
    public String getElementName();

    public String getContentType();
    public void setContentType(String contentType);

    public NamedCMNodeMap getAttributes();
    public void setAttributes(NamedCMNodeMap attributes);

}
```

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

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

public interface ElementCMModel {
    public String getListOperator();
    public void setListOperator(String listOperator);

    public int getMultiplicity();
    public void setMultiplicity(int multiplicity);

    public int getLowValue();
    public void setLowValue(int lowValue);

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

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

    public CMNodeList getDefiningElement();
    public void setDefiningElement(CMNodeList definingElement);

}
```

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

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

public interface AttributeDeclaration {
    public String getAttrName();

    public CMDataType getAttrType();
```

```
    public void setAttrType(CMDataType attrType);

    public String getDefaultValue();
    public void setDefaultValue(String defaultValue);

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

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

}
```

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

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

public interface EntityDeclaration {
}
```

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

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

import org.w3c.dom.Document;
import org.w3c.dom.CMExternalObject *;

public interface DocumentCM extends Document {
    public boolean isValid();

    public int numCMs();

    public CMObject getInternalCM();

    public CMExternalObject * getCMs();

    public CMObject getActiveCM();

    public void addCM(CMObject cm);

    public void removeCM(CMObject cm);

    public boolean activateCM(CMObject cm);

}
```

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

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

import org.w3c.dom.DomImplementation;

public interface DomImplementationCM extends DomImplementation {
    public boolean validate();
```

```
    public CMObject createCM();

    public CMExternalObject createExternalCM();

    public CMObject cloneCM(CMObject cm);

    public CMExternalObject cloneExternalCM(CMExternalObject cm);

}
```

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

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

import org.w3c.dom.how;
import org.w3c.dom.where;
import org.w3c.dom.why;
import org.w3c.dom.DOMException2;

public interface ErrorHandler {
    public void warning(where DOMString,
                        how DOMString,
                        why DOMString)
                        throws DOMException2;

    public void fatalError(where DOMString,
                           how DOMString,
                           why DOMString)
                           throws DOMException2;

    public void error(where DOMString,
                      how DOMString,
                      why DOMString)
                      throws DOMException2;

}
```

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

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

import org.w3c.dom.namedCMNodeMap;
import org.w3c.dom.nsElement;

public interface CMObject {
    public boolean getIsNamespaceAware();

    public nsElement getCMNamespace();

    public namedCMNodeMap getCMElements();

    public boolean removeCMNode(CMNode node);
```

```
    public boolean insertbeforeCMNode(CMNode newnode,
                                      CMNode parentnode);

}
```

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

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

public interface CMNode {
    public CMType getCMNodeType();

}
```

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

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

import org.w3c.dom.namedCMNodeMap;

public interface ElementDeclaration {
    public int getContentType();

    public ElementCMModel getCMElement();

    public namedCMNodeMap getCMAttributes();

    public namedCMNodeMap getCMElementsChildren();

}
```

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

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

public interface ElementCMModel {
    public ElementCMModel setCMElementCardinality(CMNode node,
                                                  int high,
                                                  int low);

    public ElementCMModel getCMElementCardinality(CMNode node,
                                                  int high,
                                                  int low);

}
```

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

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

import org.w3c.dom.Node;

public interface NodeCM extends Node {
    public boolean canInsertBefore();
```

```
    public boolean canRemoveChild();

    public boolean canReplaceChild();

    public boolean canAppendChild();

}
```

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

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

public interface ElementCMModel {
    public boolean isValid();

    public int contentType();

    public boolean canSetAttribute(String attrname,
                                   String attrval);

    public boolean canSetAttributeNode();

}
```

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

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

import org.w3c.dom.Text;

public interface TextCM extends Text {
    public boolean isWhitespaceOnly();

    public boolean canSetData();

    public boolean canAppendData();

    public boolean canReplaceData();

    public boolean canInsertData();

}
```

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

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

import org.w3c.dom.Document;

public interface DocumentCM extends Document {
    public boolean isElementDefined(String elemTypeName);

    public boolean isAttributeDefined(String elemTypeName,
                                      String attrName);
```

```
    public boolean isEntityDefined(String entName);

}
```

org/w3c/dom/contentModel/DocumentCM.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 definitions.

Object **CMObject**
Object **CMExternalObject**
Object **CMNode**
Object **CMNodeList**
Object **NamedCMNodeMap**
Object **CMDataType**
Object **CMType**
Object **ElementDeclaration**
The **ElementDeclaration** object has the following properties:
elementName
This property is of type **String**.
contentType
This property is of type **String**.
attributes
This property is of type **NamedCMNodeMap**.
Object **ElementCMModel**
The **ElementCMModel** object has the following properties:
listOperator
This property is of type **String**.
multiplicity
This property is of type **int**.
lowValue
This property is of type **int**.
highValue
This property is of type **int**.
subModels
This property is of type **NamedCMNodeMap**.
definingElement
This property is of type **CMNodeList**.
Object **AttributeDeclaration**
The **AttributeDeclaration** object has the following properties:
attrName
This property is of type **String**.
attrType
This property is of type **CMDataType**.
defaultValue
This property is of type **String**.
enumAttr
This property is of type **String**.

ownerElement
This property is of type **CMNodeList**.

Object **EntityDeclaration**

Object **DocumentCM**

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

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

isValid()
This method returns a **boolean**.

numCMs()
This method returns a **int**.

getInternalCM()
This method returns a **CMObject**.

getCMs()
This method returns a **CMExternalObject ***.

getActiveCM()
This method returns a **CMObject**.

addCM(cm)
This method has no return value.
The **cm** parameter is of type **CMObject**.

removeCM(cm)
This method has no return value.
The **cm** parameter is of type **CMObject**.

activateCM(cm)
This method returns a **boolean**.
The **cm** parameter is of type **CMObject**.

Object **DomImplementationCM**

DomImplementationCM has the all the properties and methods of **DomImplementation** as well as the properties and methods defined below.

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

validate()
This method returns a **boolean**.

createCM()
This method returns a **CMObject**.

createExternalCM()
This method returns a **CMExternalObject**.

cloneCM(cm)
This method returns a **CMObject**.
The **cm** parameter is of type **CMObject**.

cloneExternalCM(cm)
This method returns a **CMExternalObject**.
The **cm** parameter is of type **CMExternalObject**.

Object **ErrorHandler**

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

warning(DOMString, DOMString, DOMString)
This method has no return value.

The **DOMString** parameter is of type **where**.
The **DOMString** parameter is of type **how**.
The **DOMString** parameter is of type **why**.
fatalError(DOMString, DOMString, DOMString)
This method has no return value.
The **DOMString** parameter is of type **where**.
The **DOMString** parameter is of type **how**.
The **DOMString** parameter is of type **why**.
error(DOMString, DOMString, DOMString)
This method has no return value.
The **DOMString** parameter is of type **where**.
The **DOMString** parameter is of type **how**.
The **DOMString** parameter is of type **why**.
Object **CMObject**
The **CMObject** object has the following properties:
isNamespaceAware
This property is of type **boolean**.
The **CMObject** object has the following methods:
getCMNamespace()
This method returns a **nsElement**.
getCMElements()
This method returns a **namedCMNodeMap**.
removeCMNode(node)
This method returns a **boolean**.
The **node** parameter is of type **CMNode**.
insertbeforeCMNode(newnode, parentnode)
This method returns a **boolean**.
The **newnode** parameter is of type **CMNode**.
The **parentnode** parameter is of type **CMNode**.
Object **CMNode**
The **CMNode** object has the following methods:
getCMNodeType()
This method returns a **CMType**.
Object **ElementDeclaration**
The **ElementDeclaration** object has the following methods:
getContentType()
This method returns a **int**.
getCMElement()
This method returns a **ElementCMModel**.
getCMAttributes()
This method returns a **namedCMNodeMap**.
getCMElementsChildren()
This method returns a **namedCMNodeMap**.
Object **ElementCMModel**
The **ElementCMModel** object has the following methods:

setCMElementCardinality(node, high, low)
This method returns a **ElementCMModel**.
The **node** parameter is of type **CMNode**.
The **high** parameter is of type **int**.
The **low** parameter is of type **int**.
getCMElementCardinality(node, high, low)
This method returns a **ElementCMModel**.
The **node** parameter is of type **CMNode**.
The **high** parameter is of type **int**.
The **low** parameter is of type **int**.
Object **NodeCM**
NodeCM has the all the properties and methods of **Node** as well as the properties and methods defined below.
The **NodeCM** object has the following methods:
canInsertBefore()
This method returns a **boolean**.
canRemoveChild()
This method returns a **boolean**.
canReplaceChild()
This method returns a **boolean**.
canAppendChild()
This method returns a **boolean**.
Object **ElementCMModel**
The **ElementCMModel** object has the following methods:
isValid()
This method returns a **boolean**.
contentType()
This method returns a **int**.
canSetAttribute(attrname, attrval)
This method returns a **boolean**.
The **attrname** parameter is of type **String**.
The **attrval** parameter is of type **String**.
canSetAttributeNode()
This method returns a **boolean**.
Object **TextCM**
TextCM has the all the properties and methods of **Text** as well as the properties and methods defined below.
The **TextCM** object has the following methods:
isWhitespaceOnly()
This method returns a **boolean**.
canSetData()
This method returns a **boolean**.
canAppendData()
This method returns a **boolean**.
canReplaceData()
This method returns a **boolean**.

canInsertData()

This method returns a **boolean**.

Object **DocumentCM**

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

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

isElementDefined(elemTypeName)

This method returns a **boolean**.

The **elemTypeName** 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**.

isEntityDefined(entName)

This method returns a **boolean**.

The **entName** parameter is of type **String**.

Appendix C: ECMA Script Language Binding

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 at http://sisyphus.omg.org/technology/documents/formal/corba_2.htm

D.1: Normative references

Index