



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

Version 1.0

W3C Working Draft 01 November, 2000

This version:

<http://www.w3.org/TR/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20001101>
(PostScript file , PDF file , plain text , ZIP file)

Latest version:

<http://www.w3.org/TR/DOM-Level-3-Content-Models-and-Load-Save>

Previous version:

<http://www.w3.org/TR/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20000901/>

Editors:

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

Copyright © 2000 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

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 Contents3
Copyright Notice5
Chapter 1: Content Models and Validation9
Chapter 2: Document Object Model Load and Save	39
Appendix A: IDL Definitions	61
Appendix B: Java Language Binding	67
Appendix C: ECMA Script Language Binding	79
References	89
Index	91

Expanded Table of Contents

Expanded Table of Contents3
Copyright Notice5
W3C Document Copyright Notice and License5
W3C Software Copyright Notice and License6
Chapter 1: Content Models and Validation9
1.1. Overview9
1.1.1. General Characteristics9
1.1.2. Use Cases and Requirements	10
Chapter 2: Document Object Model Load and Save	39
2.1. Load and Save Requirements	39
2.1.1. General Requirements	39
2.1.2. Load Requirements	40
2.1.3. XML Writer Requirements	40
2.1.4. Other Items Under Consideration	41
2.2. Issue List	42
2.2.1. Open Issues	42
2.2.2. Resolved Issues	43
2.3. Interfaces	46
2.3.1. Interface Summary	46
2.3.2. Interfaces	47
Appendix A: IDL Definitions	61
Appendix B: Java Language Binding	67
Appendix C: ECMA Script Language Binding	79
References	89
1. Normative references	89
Index	91

Expanded Table of Contents

Copyright Notice

Copyright © 2000 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved.

This document is published under the W3C Document Copyright Notice and License [p.5] . The bindings within this document are published under the W3C Software Copyright Notice and License [p.6] . The software license requires "Notice of any changes or modifications to the W3C files, including the date changes were made." Consequently, modified versions of the DOM bindings must document that they do not conform to the W3C standard; in the case of the IDL definitions, the pragma prefix can no longer be 'w3c.org'; in the case of the Java language binding, the package names can no longer be in the 'org.w3c' package.

W3C Document Copyright Notice and License

Note: This section is a copy of the W3C Document Notice and License and could be found at <http://www.w3.org/Consortium/Legal/copyright-documents-19990405>.

Copyright © 1994-2000 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/>

Public documents on the W3C site are provided by the copyright holders under the following license. The software or Document Type Definitions (DTDs) associated with W3C specifications are governed by the Software Notice. By using and/or copying this document, or the W3C document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and distribute the contents of this document, or the W3C document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on *ALL* copies of the document, or portions thereof, that you use:

1. A link or URL to the original W3C document.
2. The pre-existing copyright notice of the original author, or if it doesn't exist, a notice of the form: "Copyright © [date-of-document] 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/>" (Hypertext is preferred, but a textual representation is permitted.)
3. *If it exists*, the STATUS of the W3C document.

When space permits, inclusion of the full text of this **NOTICE** should be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

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 © 1994-2000 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/>."
3. Notice of any changes or modifications to the W3C files, including the date changes were made. (We

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

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION 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 SOFTWARE OR DOCUMENTATION.

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

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.22]. 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*

For now, the only datatypes supported by content models will be strings; and validation will be only against string datatypes. However, if a document includes primitive datatypes supported in XML Schema, then validates against a content model, then internally, these primitive datatypes will be converted into string datatypes and then validated. The primitive datatypes supported currently are: `string`, `boolean`, `float`, `double`, `decimal`, `timeDuration`, and `recurringDuration`.

IDL Definition

```
interface CMDataType {
};
```

Interface *CMTYPE*

`CMTYPE` is a `CMNode` [p.23] 's node type. For example, one type could be `ElementDeclaration` [p.24] , composed of a tagname, content-type, etc. Others could be `CMElement` [p.25] 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.23] .

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 *CMElement*

An element in the context of a CMNode [p.23] .

IDL Definition

```

interface CMElement {
    attribute DOMString      listOperator;
    attribute CMDataType     elementType;
    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.

elementType of type CMDataType [p.13]

Datatype of 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.23] s in which the element can be defined.

Interface *AttributeDeclaration*

An attribute in the context of a CMNode [p.23] .

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 {
};

```

1.1.2.2. Validation and Other

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);
    void         setErrorHandler(in ErrorHandler handler);
};

```

Methods`activateCM`

Make the given `CObject` [p.22] 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.

Parameters

`cm` of type `CObject` [p.22]

CM to be active for the document. The `CObject` points to a list of `CExternalObject` [p.12] s; with this call, only the specified CM will be active.

Return Value

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

No Exceptions`addCM`

Associate a `CObject` [p.22] with a document. Can be invoked multiple times to result in a list of `CExternalObject` [p.12] s. Note that only one sole internal `CObject` is associated with the document, however, and that only one of the possible list of `CExternalObjects` is active at any one time.

Parameters

`cm` of type `CObject` [p.22]

CM to be associated with the document.

No Return Value**No Exceptions**`getActiveCM`

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

Return Value

`CObject` [p.22] `CObject` with a pointer to the active `CExternalObject` [p.12] of document.

No Parameters**No Exceptions**`getCMs`

Obtains a list of `CExternalObject` [p.12] s associated with a document from the `CObject` [p.22] . This list arises when `addCM()` is invoked.

Return Value

`CExternalObject` [p.12] A list of `CExternalObjects` associated with a document.

No Parameters**No Exceptions**

`getInternalCM`

Find the sole `CObject` [p.22] of a document. Only one `CObject` may be associated with the document.

Return Value

`CObject` [p.22] `CObject`.

No Parameters

No Exceptions

`isValid`

Determines if XML document is valid relative to currently active CM.

Return Value

`boolean` Valid or not.

No Parameters

No Exceptions

`numCMs`

Determines number of `CExternalObject` [p.12] s associated with the document. Only one `CObject` [p.22] can be associated with the document, but it may point to a list of `CExternalObjects`.

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

Parameters

`cm` of type `CObject` [p.22]

CM to be removed.

No Return Value

No Exceptions

`setErrorHandler`

Allow an application to register an error event handler. Where is this off of?

Parameters

`handler` of type `ErrorHandler` [p.19]

The error handler

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.22] to another `CMObject`. The `CMObject` returned wouldn't be associated with a document.

Parameters

`cm` of type `CMObject` [p.22]
`CMObject` to be cloned.

Return Value

`CMObject` [p.22] 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.22] to be cloned.

Return Value

`CMExternalObject` [p.12] Cloned `CMObject` [p.22] or NULL if failure.

No Exceptions**createCM**

Creates a `CMObject`.

Return Value

`CMObject` [p.22] 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 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 ErrorHandler {
    void          warning(in DOMLocator where,
                        in DOMString how,
                        in DOMString why)
                        raises(DOMException2);
    void          fatalError(in DOMLocator where,
                            in DOMString how,
                            in DOMString why)
                            raises(DOMException2);
    void          error(in DOMLocator where,
                       in DOMString how,
                       in DOMString why)
                       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 report conditions that are not fatal errors, and allow the calling application to continue processing.

Parameters

where of type `DOMLocator` [p.21]

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

`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` and stop all further processing.

Parameters

where of type `DOMLocator` [p.21]

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

`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 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.21]

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

DOMException2 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

1.1.2.3. CM-Editing

This section contains "CM-editing" methods (includes `CObject` [p.22] , `CMNode` [p.23] , `CMElement` [p.25] and `ElementDeclaration` [p.24] methods).

Interface *CObject*

`CObject` 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 CObject {
  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.23] list of which all `CMNodes` of type element declaration.

Return Value

`namedCMNodeMap` List of all `CMNodes` [p.23] of type element declaration.

No Parameters

No Exceptions

`getCMNamespace`

Determines namespace of `CObject`.

Return Value

`nsElement` Namespace of `CObject`.

No Parameters

No Exceptions

`insertbeforeCMNode`

Insert `CMNode` [p.23] .

Parameters

`newnode` of type `CMNode` [p.23]

`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.23] and its children, if any.

Parameters

`node` of type `CMNode` [p.23]

`CMNode` to be removed.

Return Value

`boolean` Success or failure..

No Exceptions

Interface *CMNode*

`CMNode`, or a `CObject 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 `CObject` [p.22] . 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.23] .

IDL Definition

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

Methods

`getCMAttributes`
Gets list of all attributes for this CMNode [p.23] .

Return Value

namedCMNodeMap Attributes list for this CMNode [p.23] .

No Parameters**No Exceptions**

`getCMElement`
Gets content model of element.

Return Value

CMElement [p.25] Content model of element.

No Parameters**No Exceptions**

getCMElementsChildren

Gets list of children of CMNode [p.23] .

Return Value

namedCMNodeMap Children list for this CMNode [p.23] .

No Parameters

No Exceptions

getContentType

Gets content type, e.g., empty, any, mixed, elements, PCDATA, of an element within a CMNode [p.23] .

Return Value

int Content type constant.

No Parameters

No Exceptions

Interface CMElement

An element in the context of a CMNode [p.23] .

IDL Definition

```
interface CMElement {
    CMElement      setCMElementCardinality(in CMNode node,
                                             in int high,
                                             in int low);
    CMElement      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.23]

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

CMElement Element in the context of a CMNode with its high and low values retrieved.
[p.25]

No Exceptions

setCMElementCardinality
Sets cardinality range of element's value.

Parameters

node of type CMNode [p.23]
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

CMElement [p.25]	Element in the context of a CMNode with its high and low values set.
---------------------	--

No Exceptions**1.1.2.4. Document-Editing**

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 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);
};
```

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.

Interface *ElementCM*

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

IDL Definition

```
interface ElementCM : Element {
    boolean        isValid();
    int            contentType();
    boolean        canSetAttribute(in DOMString attrname,
                                   in DOMString attrval);
    boolean        canSetAttributeNode(in Node node);
    boolean        canSetAttributeNodeNS(in Node node,
                                          in DOMString namespaceURI,
                                          in DOMString localName);
    boolean        canSetAttributeNS(in DOMString attrname,
                                     in DOMString attrval,
                                     in DOMString namespaceURI,
                                     in DOMString localName);
};
```

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

canSetAttributeNS

Determines if namespace of attribute can be set.

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 can be set.

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 namespace of attribute's node can be set.

Parameters

node of type Node

Attribute's Node in which to set the namespace.

namespaceURI of type DOMString

namespaceURI of namespace.

localName of type DOMString

localName of namespace.

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**`isValid`

Determines if the element's content is valid with respect to the currently active CM.

Return Value

`boolean` Success or failure.

No Parameters**No Exceptions****Interface *CharacterDataCM***

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

IDL Definition

```
interface CharacterDataCM : Text {
    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 {
    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 element.

`attrName` of type `DOMString`

Name of 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**1.1.2.5. 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.

1.1.2.6. 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 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.

1.1.2.7. 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.

1.1.2.8. 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 consistent. 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 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 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 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 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.

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-12:

Definition of "Non-validating". Exactly how much processing is done by "non-validing" 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.

Tentative 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-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.

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?

Issue LS-Issue-27:

How is validation handled when there are multiple possible content models associated with the document? How is one selected?

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.

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-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-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 imbed 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.

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.47] -- A new `DOMImplementation` interface that provides the factory methods for creating the objects required for loading and saving.
- `DOMBuilder` [p.47] -- A parser interface.
- `DOMInputSource` [p.53] -- Encapsulate information about the source of the XML to be loaded.
- `DOMEntityResolver` [p.54] -- During loading, provides a way for applications to redirect references to external entities.

- `DOMBuilderFilter` [p.55] -- Provide the ability to examine and optionally remove Element nodes as they are being processed during the parsing of a document.
- `DOMFormatter` [p.58] -- Provides for the actual formatting of DOM data into the output format.
- `DOMWriter` [p.56] -- An interface for writing out DOM documents. The form in which the data from the DOM will be written is controlled by a `DOMFormatter` [p.58], and the destination for the data is a `DOMOutputStream`.

2.3.2. Interfaces

Interface *DOMImplementationLS*

`DOMImplementationLS` contains the factory methods for creating objects implementing the `DOMBuilder` [p.47] (parser) and `DOMWriter` [p.56] interfaces.

IDL Definition

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

Methods

`createDOMBuilder`

Create a new `DOMBuilder` [p.47]. 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.47] The newly created parser object.

No Parameters

No Exceptions

`createDOMWriter`

Create a new `DOMWriter` [p.56] object. `DOMWriters` are used to serialize a DOM tree back into source XML form.

Return Value

`DOMWriter` [p.56] 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.47] 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

- **create-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

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.54]

If a DOMEntityResolver [p.54] 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

In the event that an error is encountered in the XML document being parsed, the DOMDocumentBuilder 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.55]

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.53] .

Parameters

is of type `DOMInputSource` [p.53]

The `DOMInputSource` from which the source document is to be read.

Return Value

Document 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` 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` 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 The newly created and populated `Document`.

Exceptions

<code>DOMException</code>	Exceptions raised by <code>parseURI()</code> originate with the installed <code>ErrorHandler</code> , and thus depend on the implementation of the <code>DOMErrorHandler</code> interfaces. The default error handlers will raise a <code>DOMException</code> if any form of XML validation or well formedness error or warning occurs during the parse, but application defined error handlers are not required to do so.
<code>DOMSystemException</code>	Exceptions raised by <code>parseURI()</code> originate with the installed <code>ErrorHandler</code> , and thus depend on the implementation of the <code>DOMErrorHandler</code> interfaces. The default error handlers will raise a <code>DOMSystemException</code> 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.55] method.

The `DOMBuilder` [p.47] 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, any encoding specified by means of this attribute will override that from the XML encoding declaration itself.

`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.47].

The `DOMBuilder` [p.47] 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.

`DOMEntityResolver` 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.47] 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.47] 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.53]

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 element is examined, it may be modified or removed, or the entire parse may be terminated early.

IDL Definition

```
interface DOMBuilderFilter {
    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****Interface *DOMWriter***

`DOMWriter` provides the API that an application will use when serializing (writing) a DOM document out in the form of a source document.

Use of a `DOMWriter` requires two other objects be supplied: a `DOMFormatter` [p.58], which defines the output format in which the document will be expressed, and a `DOMOutputStream`, which defines where the output will go.

IDL Definition

```
interface DOMWriter {
    attribute DOMFormatter    formatter;
    void    writeNode(in DOMOutputStream destination,
                    in Node node)
                raises(DOMSystemException);
    void    writeTreeWalker(in DOMOutputStream destination,
                    in TreeWalker tree)
                raises(DOMSystemException);
    void    writeString(in DOMOutputStream destination,
                    in DOMString aString)
                raises(DOMSystemException);
};
```

Attributes

`formatter` of type `DOMFormatter` [p.58]

The formatter defines the output format that will be produced when serializing using a `DOMWriter`. For now, only an XML formatter is defined, but others, such as an HTML formatter or an arbitrary user-supplied formatter, could be used.

`formatter` defaults to an XML formatter, meaning that applications do not need to explicitly set this attribute before using a `DOMWriter`.

Methods

`writeNode`

Write out the specified node, and, recursively, any children of the node. The format of the output depends on the formatter and the node type.

Parameters

`destination` of type `DOMOutputStream`

The destination for the data to be written.

`node` of type `Node`

The root node of the tree of nodes to be written.

Exceptions

<code>DOMSystemException</code>	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

`writeString`

Write out the specified `DOMString`.

Parameters

`destination` of type `DOMOutputStream`

The destination for the data to be written.

`aString` of type `DOMString`

The string to be written.

Exceptions

<code>DOMSystemException</code>	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

`writeTreeWalker`

Write out the tree of nodes selected by the specified `TreeWalker`.

Parameters

`destination` of type `DOMOutputStream`

The destination for the data to be written.

`tree` of type `TreeWalker`

The tree of nodes to be written.

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 *DOMFormatter*

`DOMFormatter` defines the interface through which the application controls the format in which a document will be written.

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.

Nodes of different types are written as follows:

- Documents are written including an XML declaration and a DTD subset, if one exists in the DOM.
- Entity nodes, when written directly by `DOMWriter.writeNode()`, output the entity expansion and a Text Decl. The resulting output will be valid as an external entity. No output is generated for any entity nodes when writing a Document.
- Entity References result in an entity reference ("`&entityName;`") in the output. Children (the expansion) of the entity reference are ignored. To write out a document with entities expanded, and no entity references, use a `TreeWalker` that is configured to deliver that kind of a view to the `DOMWriter` [p.56].
- All other node types (Element, Text, etc.) are written without modification or addition, beyond any implied by the white space or pretty printing formatting options.

Any characters that cannot be represented directly, either because of the rules of XML (& or <), or because of limitations of the output encoding, will be replaced with character references. If this is not possible (in a CDATA section, for example) the substitution character(s) will be output instead.

The XML to be written is assumed to be well formed. The output is undefined if an attempt is made to write not well formed XML, such as a Comment containing "--", or an Element containing two attributes with the same name.

Namespace prefixes, declarations and URIs are not checked for consistency by the `DOMWriter` [p.56]. If necessary, the (*there is one, right*) function from the validation module should be used to bring these items into a consistent state within the DOM prior to writing the document.

IDL Definition

```
interface DOMFormatter {
    attribute DOMString      encoding;
    readonly attribute DOMString lastEncoding;
    attribute DOMString      substituteChars;
    attribute unsigned short format;

    void formatNode(in Node rootNode,
                   in DOMOutputStream destination)
        raises(DOMSystemException);

    void formatTreeWalker(in TreeWalker tree,
                          in DOMOutputStream destination)
        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 use.
- 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.

`substituteChars` of type `DOMString`

If a character to be written can not be represented in the output encoding, the `substituteChars` string will be output in its place. If any of the characters from the `substituteChars` string can not be represented, they will be replaced by '?'. ('?' can be represented in all known character encodings.)

This substitution only occurs when serializing CDATA sections. In normal content, characters that can not be represented are output as a numeric character references.

The default value of the substitution string is "?".

Methods

`formatNode`

Format the tree whose root is the specified node, and put the resulting data to the specified output stream.

This method is not intended to be called directly from applications. Use

`DOMWriter.writeNode()` instead, which will indirectly call back here. This interface (and this method) are intended to be implemented by classes that provide alternative output

formats for DOM documents.

Parameters

rootNode of type Node

destination of type DOMOutputStream

The destination for the data to be written.

Exceptions

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

No Return Value

formatTreeWalker

Format the tree selected by the supplied TreeWalker, and put the resulting data to the specified output stream.

This method is not intended to be called directly from applications. Use DOMWriter.writeTreeWalker() instead, which will indirectly call back here. This interface (and this method) are intended to be implemented by classes that provide alternative output formats for DOM documents.

Parameters

tree of type TreeWalker

destination of type DOMOutputStream

The destination for the data to be written.

Exceptions

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

No Return Value

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/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20001101/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::int int;
    typedef dom::Node Node;
    typedef dom::nsElement nsElement;
    typedef dom::namedCMNodeMap namedCMNodeMap;
    typedef dom::Document Document;
    typedef dom::DomImplementation DomImplementation;
    typedef dom::Element Element;
    typedef dom::Text Text;
    typedef dom::DocumentType DocumentType;

    interface DOMLocator;

    interface CMObject {
    };

    interface CMExternalObject {
    };

    interface CMNode {
    };

    interface CMNodeList {
    };

    interface NamedCMNodeMap {
    };

    interface CMDataType {
    };

    interface CMType {
    };
};
```

```

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

interface CMElement {
    attribute DOMString          listOperator;
    attribute CMDataType         elementType;
    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 DOMLocator where,
                        in DOMString how,
                        in DOMString why)
                        raises(dom::DOMException2);
    void          fatalError(in DOMLocator where,
                            in DOMString how,
                            in DOMString why)
                            raises(dom::DOMException2);
    void          error(in DOMLocator where,
                       in DOMString how,
                       in DOMString why)
                       raises(dom::DOMException2);
};

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

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();
    CMElement      getCMElement();
    namedCMNodeMap getCMAttributes();
    namedCMNodeMap getCMElementsChildren();
};

interface CMElement {
    CMElement      setCMElementCardinality(in CMNode node,
                                             in int high,
                                             in int low);

    CMElement      getCMElementCardinality(in CMNode node,
                                             out int high,
                                             out int low);
};

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);
    void            setErrorHandler(in ErrorHandler handler);
};

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

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);
};

interface ElementCM : Element {

```

load-save.idl:

```
boolean    isValid();
int        contentType();
boolean    canSetAttribute(in DOMString attrname,
                          in DOMString attrval);
boolean    canSetAttributeNode(in Node node);
boolean    canSetAttributeNodeNS(in Node node,
                                  in DOMString namespaceURI,
                                  in DOMString localName);
boolean    canSetAttributeNS(in DOMString attrname,
                              in DOMString attrval,
                              in DOMString namespaceURI,
                              in DOMString localName);
};

interface CharacterDataCM : Text {
    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 {
    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);
};
};

#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::DOMString DOMString ;
    typedef dom::Element Element;
    typedef dom::DOMOutputStream DOMOutputStream;
    typedef dom::Node Node;
    typedef dom::TreeWalker TreeWalker;

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

    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;
    };
}

```

load-save.idl:

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

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

interface DOMWriter {
    attribute DOMFormatter    formatter;
    void           writeNode(in DOMOutputStream destination,
                            in Node node)
                    raises(dom::DOMSystemException);
    void           writeTreeWalker(in DOMOutputStream destination,
                                   in TreeWalker tree)
                    raises(dom::DOMSystemException);
    void           writeString(in DOMOutputStream destination,
                               in DOMString aString)
                    raises(dom::DOMSystemException);
};

interface DOMFormatter {
    attribute DOMString        encoding;
    readonly attribute DOMString    lastEncoding;
    attribute DOMString        substituteChars;
    attribute unsigned short    format;
    void           formatNode(in Node rootNode,
                             in DOMOutputStream destination)
                    raises(dom::DOMSystemException);
    void           formatTreeWalker(in TreeWalker tree,
                                   in DOMOutputStream destination)
                    raises(dom::DOMSystemException);
};
};

#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/2000/WD-DOM-Level-3-Content-Models-and-Load-Save-20001101/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/CMElement.java:

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

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

    public CMDataType getElementType();
    public void setElementType(CMDataType elementType);

    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;

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);

    public void setErrorHandler(ErrorHandler handler);
}
```

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.DOMException2;

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

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

    public void error(DOMLocator where,
                      String how,
                      String why)
        throws DOMException2;
}
```

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/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 getContentTypes();

    public CMElement getCMElement();

    public namedCMNodeMap getCMAttributes();

    public namedCMNodeMap getCMElementsChildren();
}
```

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

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

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

    public CMElement getCMElementCardinality(CMNode node,
                                             int high,
                                             int low);
}
```

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;
}
```

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

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

import org.w3c.dom.Node;
import org.w3c.dom.Element;

public interface ElementCM extends Element {
    public boolean isValid();

    public int contentType();

    public boolean canSetAttribute(String attrname,
                                   String attrval);

    public boolean canSetAttributeNode(Node node);
}
```

```
public boolean canSetAttributeNodeNS(Node node,
                                     String namespaceURI,
                                     String localName);

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

}
```

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

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

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

public interface CharacterDataCM extends Text {
    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;

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

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

}
```

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

```
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/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)
```

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

```
        throws DOMException, DOMSystemException;

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

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

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

import org.w3c.dom.DOMString ;
import org.w3c.dom.DOMReader;
import org.w3c.dom.DOMInputStream;

public interface DOMInputSource {
    public DOMInputStream getByteStream();
    public void setByteStream(DOMInputStream byteStream);

    public DOMReader getCharacterStream();
    public void setCharacterStream(DOMReader characterStream);

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

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

    public DOMString getSystemId();
    public void setSystemId(DOMString 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 endElement(Element element);
}
```

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

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

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

public interface DOMWriter {
    public DOMFormatter getFormatter();
    public void setFormatter(DOMFormatter formatter);

    public void writeNode(DOMOutputStream destination,
                          Node node)
        throws DOMSystemException;

    public void writeTreeWalker(DOMOutputStream destination,
                                TreeWalker tree)
        throws DOMSystemException;

    public void writeString(DOMOutputStream destination,
                            String aString)
        throws DOMSystemException;
}
```

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

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

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

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

    public String getLastEncoding();

    public String getSubstituteChars();
    public void setSubstituteChars(String substituteChars);

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

    public void formatNode(Node rootNode,
                           DOMOutputStream destination)
        throws DOMSystemException;
}
```

```
public void formatTreeWalker(TreeWalker tree,  
                             DOMOutputStream destination)  
    throws DOMSystemException;  
}
```

org/w3c/dom/loadSave/DOMFormatter.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

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 read-only property is of type **String**.

contentType

This property is of type **String**.

attributes

This property is a **NamedCMNodeMap** object.

Object **CMElement**

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

listOperator

This property is of type **String**.

elementType

This property is a **CMDataType** object.

multiplicity

This property is a **int** object.

lowValue

This property is a **int** object.

highValue

This property is a **int** object.

subModels

This property is a **NamedCMNodeMap** object.

definingElement

This property is a **CMNodeList** object.

Object **AttributeDeclaration**

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

attrName

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

attrType

This property is a **CMDataType** object.

defaultValue

This property is of type **String**.

enumAttr

This property is of type **String**.

ownerElement

This property is a **CMNodeList** object.

Object **EntityDeclaration**

Object

Object **DocumentCM**

DocumentCM has all the properties and methods of the **Document** object 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** object.

getInternalCM()

This method returns a **CObject** object.

getCMs()

This method returns a **CMExternalObject** object.

getActiveCM()

This method returns a **CObject** object.

addCM(cm)

This method has no return value.

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

removeCM(cm)

This method has no return value.

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

activateCM(cm)

This method returns a **Boolean**.

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

setErrorHandler(handler)

This method has no return value.

The **handler** parameter is a **ErrorHandler** object.

Object **DomImplementationCM**

DomImplementationCM has 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:

validate()

This method returns a **Boolean**.

createCM()

This method returns a **CObject** object.

createExternalCM()

This method returns a **CMExternalObject** object.

cloneCM(cm)

This method returns a **CObject** object.

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

cloneExternalCM(cm)

This method returns a **CMExternalObject** object.

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

Object **ErrorHandler**

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

warning(when, how, why)

This method has no return value.

The **when** parameter is a **DOMLocator** object.

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

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

This method can raise a **DOMException2** object.

fatalError(when, how, why)

This method has no return value.

The **when** parameter is a **DOMLocator** object.

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

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

This method can raise a **DOMException2** object.

error(when, how, why)

This method has no return value.

The **when** parameter is a **DOMLocator** object.

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

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

This method can raise a **DOMException2** 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

Object **CMObject**

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

isNamespaceAware

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

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

getCMNamespace()

This method returns a **nsElement** object.

getCMElements()

This method returns a **namedCMNodeMap** object.

removeCMNode(node)

This method returns a **Boolean**.

The **node** parameter is a **CMNode** object.

insertbeforeCMNode(newnode, parentnode)

This method returns a **Boolean**.

The **newnode** parameter is a **CMNode** object.

The **parentnode** parameter is a **CMNode** object.

Object **CMNode**

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

getCMNodeType()

This method returns a **CMType** object.

Object **ElementDeclaration**

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

getContentTypes()

This method returns a **int** object.

getCMElement()

This method returns a **CMElement** object.

getCMAttributes()

This method returns a **namedCMNodeMap** object.

getCMElementsChildren()

This method returns a **namedCMNodeMap** object.

Object **CMElement**

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

setCMElementCardinality(node, high, low)

This method returns a **CMElement** object.

The **node** parameter is a **CMNode** object.

The **high** parameter is a **int** object.

The **low** parameter is a **int** object.

getCMElementCardinality(node, high, low)

This method returns a **CMElement** object.

The **node** parameter is a **CMNode** object.

The **high** parameter is a **int** object.

The **low** parameter is a **int** object.

Object

Object **NodeCM**

NodeCM has 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.

Object **ElementCM**

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

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

isValid()

This method returns a **Boolean**.

contentType()

This method returns a **int** 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, namespaceURI, localName)

This method returns a **Boolean**.

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

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

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

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**.

Object **CharacterDataCM**

CharacterDataCM has the all the properties and methods of the **Text** 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** 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 **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 a **DOMString** object.

publicId

This property is a **DOMString** object.

systemId

This property is a **DOMString** object.

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:

endElement(element)

This method returns a **Boolean**.

The **element** parameter is a **Element** object.

Object **DOMWriter**

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

formatter

This property is a **DOMFormatter** object.

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.

writeTreeWalker(destination, tree)

This method has no return value.

The **destination** parameter is a **DOMOutputStream** object.

The **tree** parameter is a **TreeWalker** object.

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

writeString(destination, aString)

This method has no return value.

The **destination** parameter is a **DOMOutputStream** object.

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

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

Object **DOMFormatter**

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

encoding

This property is of type **String**.

lastEncoding

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

substituteChars

This property is of type **String**.

format

This property is of type **Number**.

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

formatNode(rootNode, destination)

This method has no return value.

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

The **destination** parameter is a **DOMOutputStream** object.

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

formatTreeWalker(tree, destination)

This method has no return value.

The **tree** parameter is a **TreeWalker** object.

The **destination** parameter is a **DOMOutputStream** object.

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

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

activateCM	addCM	AttributeDeclaration
attributes	attrName	attrType
byteStream		
canAppendChild	canAppendData	canDeleteData
canInsertBefore	canInsertData	canRemoveChild
canReplaceChild	canReplaceData	canSetAttribute
canSetAttributeNode	canSetAttributeNodeNS	canSetAttributeNS
canSetData	canSetFeature	CharacterDataCM
characterStream	cloneCM	cloneExternalCM
CMDataType	CMElement 14, 25	CMExternalObject
CMNode 12, 23	CMNodeList	CMObject 12, 22
CMTType	contentType 29, 14	createCM
createDOMBuilder	createDOMWriter	createExternalCM
defaultValue	definingElement	DocumentCM
DocumentTypeCM	DOMBuilder	DOMBuilderFilter
DOMEntityResolver	DOMFormatter	DomImplementationCM
DOMImplementationLS	DOMInputSource	DOMLocator
DOMWriter		
ECMAScript	ElementCM	ElementDeclaration 13, 24
elementName	elementType	encoding 54, 59
endElement	EntityDeclaration	entityResolver
enumAttr	error	ErrorHandler 19, 49

fatalError	filter	format
formatNode	formatter	formatTreeWalker
getActiveCM	getCMAttributes	getCMElement
getCMElementCardinality	getCMElements	getCMElementsChildren
getCMNamespace	getCMNodeType	getCMs
getColumnNumber	getContentType	getFeature
getInternalCM	getLineNumber	getNode
getPublicID	getSystemID	
highValue		
insertbeforeCMNode	isAttributeDefined	isAttributeDefinedNS
isElementDefined	isElementDefinedNS	isEntityDefined
isNamespaceAware	isValid 17, 30	isWhitespaceOnly
Java		
lastEncoding	listOperator	lowValue
multiplicity		
NamedCMNodeMap	NodeCM	numCMs
OMGIDL	ownerElement	
parseDOMInputSource	parseURI	publicId

removeCM

removeCMNode

resolveEntity

setCMElementCardinality

setErrorHandler

setFeature

subModels

substituteChars

supportsFeature

systemId

validate

warning

writeNode

writeString

writeTreeWalker