WD-DOM-19980416

Document Object Model Specification

Version 1.0

WD-DOM-19980416

World Wide Web Consortium

16 April, 1998

This version

http://www.w3.org/TR/1998/WD-DOM-19980416
http://www.w3.org/TR/1998/WD-DOM-19980416/DOM.ps
http://www.w3.org/TR/1998/WD-DOM-19980416/DOM.tgz
http://www.w3.org/TR/1998/WD-DOM-19980416/DOM.zip

Previous versions

http://www.w3.org/TR/WD-DOM-19980318
http://www.w3.org/TR/WD-DOM-971209
http://www.w3.org/TR/WD-DOM-971009

Latest version

http://www.w3.org/TR/WD-DOM

Status

This is a W3C Working Draft for review by W3C members and other interested parties. 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". This is work in progress and does not imply endorsement by, or the consensus of, either W3C or members of the DOM working group.

This document has been produced as part of the W3C DOM Activity , and is intended as a draft of a proposed recommendation for the Document Object Model. The authors of this document are the DOM WG members. Different chapters of the Document Object Model specification may have different editors.

We will update this draft specification on a regular basis.

Please send detailed comments on this document to www-dom@w3.org . We cannot guarantee a personal response but we will try when it is appropriate. Public discussion about the DOM also takes place on this mailing list.

Abstract

This specification defines the Document Object Model, a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. Vendors can support the DOM as an interface to their proprietary data structures and APIs, and content authors can write to the standard DOM interfaces rather than product-specific APIs, thus increasing interoperability on the Web.

The goal of the DOM specification is to define a programmatic interface for XML and HTML. The DOM specification is separated into three parts: Core, HTML, and XML. The Core DOM specification provides a low-level set of objects that can represent any structured document. The core interface is a compact and minimal design for manipulating the document's contents, capable of representing any HTML or XML document. Depending upon the DOM's usage, the core DOM interface may not be convenient or appropriate for all users. The HTML and XML specifications provide additional, higher-level interfaces that are used with the core specification to provide a more convenient view into the document. These specifications consist of objects and methods that provide easier and more direct access into the specific types of documents.

WG Chair

Lauren Wood, *SoftQuad, Inc.*

Editors

Vidur Apparao, *Netscape*
Steve Byrne, *JavaSoft*
Mike Champion, *ArborText*
Scott Isaacs, *Microsoft*
Gavin Nicol, *Inso EPS*
Jared Sorensen, *Novell*
Robert Sutor, *IBM*
Chris Wilson, *Microsoft*
Lauren Wood, *SoftQuad, Inc.*

Principal Contributors

Vidur Apparao, *Netscape*
Steve Byrne, *JavaSoft (until November 1997)*
Mike Champion, *ArborText, Inc.*
Scott Isaacs, *Microsoft (until January, 1998)*
Arnaud Le Hors, *W3C*
Gavin Nicol, *Inso EPS*
Jonathan Robie, *Texcel*
Peter Sharpe, *SoftQuad, Inc*
Bill Smith, *Sun (from November 1997)*
Jared Sorensen, *Novell*
Robert Sutor, *IBM*
Chris Wilson, *Microsoft (after January, 1998)*

Table of contents

Table of contents

Document Object Model Specification

Expanded table of contents

1. Document Object Model Requirements

Editors

Jared Sorensen, *Novell, Inc.*
Lauren Wood, *SoftQuad, Inc.*

This document defines the high-level requirements for the Document Object Model (DOM). It is part of the Document Object Model Specification .

1.1. General Requirements

1. References to XML and HTML documents generally denote the physical files that contain structural markup.
2. Some requirements are not implemented in DOM Level 1. Those requirements are identified in this document with the notation *[After Level 1]*.

Listed below are the general requirements of the Document Object Model.

1. The Object Model is language neutral and platform independent.
2. There will be a core DOM that is applicable to HTML, CSS and XML documents.
3. The Object Model can be used to construct and deconstruct the document.
4. The Object Model will not preclude use by either agents external to the document content, or scripts embedded within the document.
5. Consistent naming conventions must be used through all levels of the Object Model.
6. A visual UI component will not be required for a conforming implementation of the Object Model.
7. The specific HTML, CSS or XML document object models will be driven by the underlying constructs of those languages.
8. It must be possible to read in a document and write out a structurally isomorphic document.
9. The Object Model will not expose the user to problems with security, validity, or privacy.
10. The Object Model will not preclude other mechanisms for manipulating documents.

1.2. Structure Navigation

This refers to the navigation around a document, such as finding the parent of a given element, or what children elements a given parent element contains.

1.2.1. General Requirements

1. All document content, including elements and attributes, will be programmatically accessible and manipulable.
2. Navigation from any element to any other element will be possible, except where such navigation

would compromise security.

3. There will be a way to uniquely and reproducibly enumerate the structure of static documents.
4. There will be a way to query for elements and attributes, subject to security and privacy considerations.
5. Basic low-level functions (get first, get next, etc.) will be provided, along with convenience functions that build upon them, but have a consistent access method.

1.2.2. HTML Requirements

These are specific to HTML documents.

1. All elements known to the user agent are exposed.
2. Unknown tags and attributes are exposed.
3. Implied elements are exposed even if not explicitly defined in the document (e.g., HTML, HEAD, BODY).
4. There will be guidelines for the inclusion of new elements and attributes in the object model.

1.3. Document Manipulation

1. There will be a way to add, remove and change elements and/or tags (if permitted by the Document Type Definition and not precluded by security or validity considerations) in the document structure.
2. There will be a way to add, remove and change attributes (if permitted by the Document Type Definition and not precluded by security or validity considerations) in the document structure.
3. Operations (or a combination of operations) must restore consistency before they return.
4. A valid static document acted upon by the DOM will deliver a consistent reproducible document structure.

1.4. Content Manipulation

1. There will be a way to determine the containing element from any text part of the document (subject to security considerations).
2. There will be a way to manipulate (add, change, delete) content.
3. There will be a way to navigate content.

1.5. Event Model *[After Level 1]*

The event model must be rich enough to create completely interactive documents. This requires the ability to respond to any user action that may occur on the document. Therefore, many of these requirements only apply if a UI component is involved.

1. All elements will be capable of generating events.
2. There will be interaction events, update events, and change events.
3. The event model will allow responses to user interactions.

4. The event delivery mechanism will allow for overriding of default behavior.
5. The event model will provide a mechanism by which events for specific elements may be received by an ancestor in the DOM hierarchy.
6. Events are synchronous.
7. Events will be defined in a platform independent and language neutral way.
8. There will be an interface for binding to events.

1.6. Stylesheet Object Model *[After Level 1]*

Cascading Style Sheets (CSS) is one model for manipulating the style of the document. The Stylesheet Object Model exposes the ability to create, modify, and associate CSS style sheets with the document. The stylesheet model will be extensible to other stylesheet formats in the future.

1. All style sheets will be represented in the object model.
2. There will be a CSS stylesheet model. The CSS object model will be defined as part of a stylesheet embedding model, where the core part of the model may be applicable to other style languages.
3. Selectors, rules and properties of individual style sheets can be added, removed and changed.
4. All elements of a CSS style can be added, removed, and changed in the object model. This includes but is not limited to:
    1. linked style sheets
    2. imported style sheets
    3. alternative style sheets
    4. CSS pseudo-classes and CSS pseudo-elements
    5. contextual selectors
    6. inline styles
    7. all properties as defined in the CSS specification, including but not limited to font properties, colors, backgrounds, and box properties.

1.7. DTD Manipulation

1. *[After Level 1]* There will be a way to determine the presence of a DTD.
2. There will be a way to query declarations in the underlying DTD (if available).
3. *[After Level 1]* There will be a way to add, remove, and change declarations in the underlying DTD (if available).
4. *[After Level 1]* There will be a way to test conformance of all or part of the given document against a DTD (if available). (See Document Manipulation, item 3).

1.8. Error Reporting

1. The DOM will provide a document-wide error logging and reporting mechanism.
2. Error reporting will be primarily via exceptions.
3. The DOM error state can be queried.

1.9. Security, Validity and Privacy *[After Level 1]*

Security, validity, and privacy considerations are interrelated and entwined. DOM will initially provide simple "sandbox" security; subsequent levels are expected to incorporate more sophisticated mechanisms.

1. Each object must be responsible for maintaining its own internal consistency.
2. It must be safe to have multiple threads operating on the same object.
3. Object locking must be incorporated to ensure consistent results.
4. It must be possible to prevent scripts on one page from accessing another page.
5. Firewall boundaries must be respected.
6. It must be possible to restrict access and navigation to specific elements.
7. An external security API will be provided.

1.10. Document Meta Information

These are requirements for information about the document.

1. There will be a way of obtaining relevant information about the document and its embedded objects such as source location, date created, and associated cookies.

1.11. UA Information

These are requirements for information about the user agent environment, if applicable.

1. There will be a way of obtaining relevant information about the display environment, including the UA brand information and version number, and, where appropriate, the HTTP header.
2. *[After Level 1]* A way of determining support for a MIME type will be available.

2. Document Object Model (Core) Level 1

Editors

Mike Champion, *ArborText (from November 20, 1997)*
Steve Byrne, *JavaSoft (until November 19, 1997)*

Level One of the Document Object Model (DOM) level one provides a common API for software developers and web script authors to access and manipulate parsed HTML and XML content inside conforming products. All markup as well as any document type declarations are made available. Level one also allows creation "from scratch" of entire web documents in memory; saving those documents persistently is left to the programmer. DOM Level one is intentionally limited in scope to methods to represent and manipulate document structure and content; a standard API for controlling how documents are rendered via stylesheets, validated against a DTD, accessed by given users, etc. will be specified in a subsequent revision level of the DOM.

2.1. Introduction

This specification defines a minimal set of objects and interfaces for accessing and manipulating document objects. The functionality specified in this draft (the "*core*" functionality) should be sufficient to implement higher level operations, such as querying, and filtering of a document; future drafts will add "convenience" operations which can be implemented in terms of the *core* operations, but which may be implemented more efficiently using implementation-specific mechanisms that fall outside of the scope of this specification.

2.2. Intended Audience

The audience for this document is intended to be web script authors, software developers, and DOM implementation providers. This document presumes familiarity with concepts and terminology from HTML and XML, as well as object oriented programming. The actual DOM specification is provided in the Object Management Group's Interface Definition Language (IDL); experience with Java or C++ syntax should be sufficient to allow comprehension.

2.3. Design goals for the DOM Level One specification

It is important to note that the DOM is an "abstraction", or an idealized model of how documents are represented and manipulated in the products that support the DOM interfaces. Or, even more ideally, one might think of the DOM as the "abstract base classes" that products supporting the DOM actually implement. Thus, in general, HTML and XML products that support the DOM merely "expose" the DOM as a means of accessing and manipulating their (possibly proprietary) internal data structures and operations; there is no assumption that they are internally built on the DOM APIs or that they do not expose proprietary APIs that have no relationship to the W3C DOM.

The DOM objects and interfaces are designed to be:

- high-fidelity: sufficient for representing the content of parsed HTML and XML documents without loss of [significant] information. The supported HTML version is 4.0; the supported XML version is 1.0.
- isomorphic with XML: The DOM should be sufficient to construct an entirely new document instance programmatically that is identical to the parsed form of a given HTML or XML document. This means that it has sufficient constructive power to build any useful document object hierarchy, and that an implementation could be written such that the external document parser merely calls the methods specified in the level one specification to build the object hierarchy.
- extensible: The DOM core is the foundation for the rest of the document object model levels, which means it must be simple, flexible, and extensible.
- thread-safe: The operations supported by the DOM will not corrupt the document object or return corrupted state (as far as this API is concerned). Higher level consistency support mechanisms such as explicit locks or transactions are outside of the scope of the level one specification. For level one of the DOM, the assumption is that only one thread operates on the document at a time.

Note: *In the current specification, some operations can modify the document tree, but there is no model for handling concurrent access. The WG also recognises that in some situations, a document, or some of its components, will not be modifiable, and a method for dealing with such situations needs to be defined in a subsequent revision level of the DOM.*

2.3.1. Entities and the DOM Core

In the DOM core, there are no objects representing entities. Numeric character references and references to the pre-defined entities in HTML and XML, are replaced by the single character that makes up the entitity's replacement. For example, in:

```
<p>This is a dog &amp; a cat</p>
```

the "&" will be replaced by the character "&", and the text in the <p> element will form a single continuous sequence of characters. The representation of general entities, both internal and external, are defined within the XML-specific portion of the level one specification.***Note:*** *When a DOM representation of a document is serialized as XML or HTML text, applications will need to check each character in text data to see if it needs to be escaped using a numeric or pre-defined entity. Failing to do so could result in invalid HTML or XML.*

2.4. IDL Issues

The primary Document Object Model type definitions are presented using the Object Management Group's Interface Definition Language (IDL, ISO standard 14750). While a complete tutorial on the IDL language is beyond the scope of this document, a few key items deserve explicit mentioning:

- In the DOM’s IDL definition, within each interface there are primarily two kinds of things:
  1. Attribute definitions
  2. Method (IDL calls them "operation") definitions
- Attribute definitions are a *shorthand* notation for a pair of "get/set" or accessor/mutator methods, and should *not* be thought of, nor mapped in a particular language binding to, data members directly.
- The "constructor" methods for the various IDL objects are not specified explicitly. Specific programming language implementations of the DOM will provide suitable object creation methods; typically these will be simple with no argument constructors, although this is not a requirement. For the most part, DOM users will call the createXXX() methods on the Document class to create instances of DOM objects that are associated with a given document.
- IDL’s `long` datatype represents 32 signed bit integers. In other language bindings, for example Java, this would be mapped to the Java `int` datatype.

More information on IDL is available from OMG and in chapter 3 of the CORBA 2.0 specification.***Note:*** *The Object Management Group Interface Definition Language (OMG IDL) was chosen as it was designed for specifying language and implementation-neutral interfaces. Various other IDLs could be used; the use of OMG IDL does not imply a requirement to use a specific object binding runtime.*

2.5. Primary API Types

The types described in this section are those that application programmers using the DOM will encounter most frequently. A good working knowledge of these types will be sufficient to accomplish most tasks.

2.5.1. Node

Node is the base type of most objects in the Document Object Model. It may have an arbitrary number (including zero) of sequentially ordered child nodes. It usually has a parent Node; the exception being that the root Node in a document hierarchy has no parent.

2.5.2. Element

Element objects represent the elements in the HTML or XML document. Elements contain, as child nodes, all the content between the start tag, and the end tag of an element. Additionally, Element objects have a list of Attribute objects which represent the combination of those attributes explicitly specified in the document, and those defined in the document type definition which have default values.

2.5.3. DocFragment

The DocFragment object represents the root of a lightweight document or document fragment.

2.5.4. Document

The Document object extends the DocFragment interface to represent the root node of a standalone document.

2.6. Auxiliary types

These types are "helpers" which can appear in various parts of the DOM. Some of them occur quite frequently in common usage; others are limited to the Document Type Definition section of the document, and thus may be of little interest to typical DOM users. Those types are marked accordingly.

2.6.1. NodeIterator

The NodeIterator object is used for iterating over a set of nodes specified by a "filter". Iterators are described in detail in the book *Design Patterns*, pages 257-271, from which the following sections are quoted:

Intent
: Provide a way to access elements of an aggregate object sequentially without exposing it's underlying representation.

Applicability
: Use the Iterator pattern

- to access an aggregate object's content without exposing it's internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

2.6.2. AttributeList

The AttributeList object represents a collection of Attribute objects, indexed by attribute name.

2.6.3. DocumentContext

The DocumentContext object is repository for meta-data about a document, such as source, creation date, and other creation context information. This object will be fully specified in the level two DOM specification.

2.6.4. DOM

The DOM object provides document basic operations such as creating new top-level HTML or XML documents, or making inquiries about specific supported versions of HTML and XML within a particular document object model implementation.

2.6.5. Range

The Range object *[Being discussed for possible inclusion in a later draft of level one]* represents a part of the document, potentially including markup.

2.7. Typical operation

Normally, a DOM-compliant implementation will make the main Document instance available to the application through some implementation-specific mechanism. For example, a typical implementation would pass the application a reference to a `DocumentContext` object. From the `DocumentContext`, the application may retrieve the `Document` object, which is the root of the document object hierarchy.

Once the application has access to the root of the document object hierarchy, it can use the methods defined herein for accessing individual nodes, selection of specific node types such as all images, and so on.

2.8. Document Object Model APIs

This section defines the complete set of objects and methods which are defined by the Document Object Model. The general structure of these object definitions is:

- A brief overview of the semantics of the object and how it relates to other objects in the DOM.
- A sequence of <method signature, description> pairs for each method that is defined for the object.

Interface *DOM*

The "DOM" interface provides a number of methods for performing operations that are independent of any particular instance of the document object model. Although IDL does not provide a mechanism for expressing the concept, the methods supplied by the DOM interface will be implemented as "static", or instance independent, methods. This means that a client application using the DOM does not have to locate a specific instance of the DOM object; rather, the methods are will be available directly on the DOM class itself and so are directly accessible from any execution context.

IDL Definition

```
interface DOM {
  Document             createDocument(in wstring type);
  boolean              hasFeature(in wstring feature);
};
```

Method `createDocument()`

Creates a document of the specified type

Parameters

type	The type of document to create, i.e., "HTML" or "XML".

Return Values

Document	An instance of an HTML or XML document is created. Documents are a special type of DOM object because other objects (such as Elements, DocumentFragments, etc.) can only exist within the context of a Document.

Exceptions

This method throws no exceptions.

Method `hasFeature()`

Returns TRUE if the current DOM implements a given feature, FALSE otherwise.

Parameters

feature	The package name of the feature to test.

Return Values

boolean	

Exceptions

This method throws no exceptions.

Interface *DocumentContext*

The DocumentContext object represents information that is not strictly related to a document's content; rather, it provides the information about where the document came from, and any additional meta-data about the document. For example, the DocumentContext for a document retrieved using HTTP would provide access to the HTTP headers which were retrieved with the document, the URL that the document came from, etc.

For documents which were not retrieved via HTTP, or for those which were created directly in memory, there may be no DocumentContext.***Note:*** *The DocumentContext interface described here is expected to be significantly expanded in the level two specification of the Document Object Model.*

IDL Definition

```
interface DocumentContext {
  attribute Document       document;
};
```

Attribute `document`

This is the root node of a Document Object Model. Any iteration, enumeration or other traversal of the entire document's content should begin with this node.

Interface *DocumentFragment*

DocumentFragment is the "lightweight" or "minimal" document object, or the handle for a "fragment" as returned by the Range operations, and it (as the superclass of Document) anchors the XML/HTML tree in a full-fledged document.

The children of a DocumentFragment node are zero or more nodes representing the tops of any sub-trees defining the structure of the document, including the actual root element of the XML or HTML document, as well as the XML prolog, comments, PIs, etc. For a document fragment, there could be a number of subtrees, since fragments do not need to be well-formed XML documents (although they do need to be well-formed XML parsed entities, which can have multiple top nodes). Criteria for "well-formedness" are much looser in the HTML world, and the DOM will not attempt to impose any constraints here. For a complete HTML document, this will contain an Element instance whose tagName is `"HTML"`; for a complete XML document this will contain the outermost element, i.e. the element non-terminal in production *[41]* in Section 3 of the XML-lang specification.

IDL Definition

```
interface DocumentFragment : Node {
  attribute Document       masterDoc;
};
```

Attribute ***masterDoc***

This provides access to the Document object asssociated with this DocumentFragment.

Interface *Document*

The Document object represents the entire HTML or XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data. Since Document inherits from DocumentFragment, its children are contents of the Document, e.g., the root Element, the XML prolog, any processing instructions and or comments, etc.

Since Elements, Text nodes, Comments, PIs, etc. cannot exist outside a Document, the Document interface also anchors the "factory" methods that create these objects.

IDL Definition

```
interface Document : DocumentFragment {
  attribute Node            documentType;
  attribute Element         documentElement;
  attribute DocumentContextcontextInfo;
  DocumentContext     createDocumentContext();
  Element             createElement(in wstring tagName,
                                    in AttributeList attributes);
  Text                createTextNode(in wstring data);
  Comment             createComment(in wstring data);
  PI                  createPI(in wstring name,
                               in wstring data);
  Attribute           createAttribute(in wstring name,
                                      in Node value);
  AttributeList       createAttributeList();
  TreeIterator        createTreeIterator(in Node node);
  NodeIterator        getElementsByTagName(in wstring tagname);
};
```

Attribute ***documentType***

For XML, this provides access to the Document Type Definition (see DocumentType) associated with this XML document. For HTML documents and XML documents without a document type definition this returns the value `null`.

Attribute ***documentElement***

This is a "convenience" attribute to jump directly to the child node that is the root element of the document.

Attribute *`contextInfo`*

Detailed information on where this document came from and any additional information we need to define.

Method *`createDocumentContext()`*

Create and return a new DocumentContext.

Parameters

This method has no parameters.

Return Values

DocumentContext	A new `DocumentContext` object.

Exceptions

This method throws no exceptions.

Method *`createElement()`*

Create an element based on the tagName. Note that the instance returned may implement an interface derived from `Element`. The attributes parameter can be null if no attributes are specified for the new `Element`.

Parameters

tagName	The name of the element type to instantiate.
attributes	The set of attributes for the element.

Return Values

Element	A new `Element` object with a set of attributes identical to those spoecified in the parametert list.

Exceptions

This method throws no exceptions.

Method `createTextNode()`

Create a `Text` node given the specified string.

Parameters

data	The data for the node.

Return Values

Text	The new `Text` object.

Exceptions

This method throws no exceptions.

Method `createComment()`

Create a `Comment` node given the specified string.

Parameters

data	The data for the node.

Return Values

Comment	The new `Comment` object.

Exceptions

This method throws no exceptions.

Method *createPI()*

Create a `PI` node given the specified name and data strings.

Parameters

name	The name part of the PI.
data	The data for the node.

Return Values

PI	The new `PI` object.

Exceptions

This method throws no exceptions.

Method *createAttribute()*

Create an Attribute of the given name and specified value. Note that the Attribute instance can then be set on an Element using the setAttribute method.

Parameters

name	The name of the attribute.
value	The value of the attribute.

Return Values

Attribute	A new `Attribute` object.

Exceptions

This method throws no exceptions.

Method *`createAttributeList()`*

Create an empty AttributeList.

Parameters

This method has no parameters.

Return Values

AttributeList	A new `AttributeList` object.

Exceptions

This method throws no exceptions.

Method *`createTreeIterator()`*

Create a TreeIterator anchored on a given node.

Parameters

node	The "root" node of this tree iterator.

Return Values

TreeIterator	A new `TreeIterator` object.

Exceptions

This method throws no exceptions.

Method *getElementsByTagName()*

Returns an iterator through all subordinate Elements with a given tag name.

Parameters

tagname	The name of the tag to match on

Return Values

NodeIterator	A new `NodeIterator` object.

Exceptions

This method throws no exceptions.

Interface *Node*

The Node object is the primary datatype for the entire Document Object Model. It represents a single node in the document tree. Nodes may have, but are not required to have, an arbitrary number of child nodes.

IDL Definition

```
interface Node {
  // NodeType
  const int            DOCUMENT             = 1;
  const int            ELEMENT              = 2;
  const int            ATTRIBUTE            = 3;
  const int            PI                   = 4;
  const int            COMMENT              = 5;
  const int            TEXT                 = 6;

  int                 getNodeType();
  Node                getParentNode();
  NodeIterator        getChildNodes();
  boolean             hasChildNodes();
  Node                getFirstChild();
  Node                getPreviousSibling();
  Node                getNextSibling();
  Node                insertBefore(in Node newChild,
                                   in Node refChild)  interface NotMyChildException {};
;
  Node                replaceChild(in Node newChild,
                                   in Node oldChild)  interface NotMyChildException {};
;
  Node                removeChild(in Node oldChild)  interface NotMyChildException {};
;
};
```

Definition group *NodeType*

Defined Constants

DOCUMENT	
ELEMENT	
ATTRIBUTE	
PI	
COMMENT	
TEXT	

Method *getNodeType()*

Returns an indication of the underlying Node object's type. The actual type of the returned data is language binding dependent; the IDL specification uses an `enum`, and it is expected that most language bindings will represent this runtime-queryable Node type using an integral data type. The names of the node type enumeration literals are straightforwardly derived from the names of the actual Node subtypes, and are fully specified in the IDL definition of Node in the IDL definition in Appendix A.

Parameters

This method has no parameters.

Return Values

int	The type of the node.

Exceptions

This method throws no exceptions.

Method *getParentNode()*

Returns the parent of the given Node instance. If this node is the root of the document object tree, or if the node has not been added to a document tree, `null` is returned. [Note: because in ECMAScript get/set method pairs are surfaced as properties, Parent would conflict with the pre-defined Parent property, so we disambiguate this with "ParentNode" even though it is inconsistent with the naming convention of the other methods that do not include "Node"].

Parameters

This method has no parameters.

Return Values

Node	The parent of the node.

Exceptions

This method throws no exceptions.

Method `getChildNodes()`

Returns a NodeIterator object that will enumerate all children of this node. If there are no children, an iterator that will return no nodes is returned. The content of the returned NodeIterator is "live" in the sense that changes to the children of the Node object that it was created from will be immediately reflected in the nodes returned by the iterator; it is not a static snapshot of the content of the Node. Similarly, changes made to the nodes returned by the iterator will be immediately reflected in the tree, including the set of children of the Node that the NodeIterator was created from.

Parameters

This method has no parameters.

Return Values

NodeIterator	An iterator through the children of the node.

Exceptions

This method throws no exceptions.

Method `hasChildNodes()`

Returns `true` if the node has any children, `false` if the node has no children at all. This method exists both for convenience as well as to allow implementations to be able to bypass object allocation, which may be required for implementing `getChildNodes()`.

Parameters

This method has no parameters.

Return Values

boolean	True if the node has children.

Exceptions

This method throws no exceptions.

Method *`getFirstChild()`*

Returns the first child of a node. If there is no such node, `null` is returned.

Parameters

This method has no parameters.

Return Values

Node	The first child of the node, or null.

Exceptions

This method throws no exceptions.

Method *`getPreviousSibling()`*

Returns the node immediately preceding the current node in a breadth-first traversal of the tree. If there is no such node, `null` is returned.

Parameters

This method has no parameters.

Return Values

Node	The the node immediately preceeding, or null.

Exceptions

This method throws no exceptions.

Method `getNextSibling()`

Returns the node immediately following the current node in a breadth-first traversal of the tree. If there is no such node, `null` is returned.

Parameters

This method has no parameters.

Return Values

Node	The node immediately following, or null.

Exceptions

This method throws no exceptions.

Method `insertBefore()`

Inserts a child node (newChildbefore the existing child node refChild. If refChild is `null`, insert newChild at the end of the list of children. If refChild is not a child of the Node that `insertBefore` is being invoked on, a `NotMyChildException` is thrown.

Parameters

newChild	
refChild	

Return Values

Node	The node being inserted.

Exceptions

NotMyChildException	Thrown if `refChild` is not a child of the node.

Method *replaceChild()*

Replaces the child node oldChild with newChild in the set of children of the given node, and return the oldChild node. If oldChild was not already a child of the node that the `replaceChild` method is being invoked on, a `NotMyChildException` is thrown.

Parameters

newChild	
oldChild	

Return Values

Node	The node being replaced.

Exceptions

NotMyChildException	Thrown if `oldChild` is not a child of the node.

Method *removeChild()*

Removes the child node indicated by oldChild from the list of children and returns it. If oldChild was not a child of the given node, a `NotMyChildException` is thrown.

Parameters

oldChild	

Return Values

Node	The node being removed.

Exceptions

NotMyChildException	Thrown if `oldChild` is not a child of the node.

Interface *NodeIterator*

A NodeIterator is a simple iterator interface that can be used to provide a linear view of the document hierarchy. The NodeIterator is similar in concept to a the Java "Enumeration" interface, but has a much richer set of methods: Java enumerators objects implementing the java.util.Enumeration interface) iterate across a set of Objects; the first element in the set is retrieved by the first call to a nextElement() method, and subsequent calls to nextElement() return each dditional element in the set in succession. A boolean hasMoreElements() method may be called to determine whether there are more elements in the set that have not been returned yet.

The DOM NodeIterator interface allows the same basic operation: The first Node in a set of nodes defined by a NodeIterator is returned by a call to the getNextNode() method, and subsequent calls to getNextNode() will return the rest of the set. The NodeIterator, however, lets the caller re-position the iterator to the beginning or the end of the set of Nodes, to determine how many members of the set there are, position the iterator to an arbitrary point in the set (either by position or by looking up a particular Node), to get the current position, and to iterate either forwards or backwards through the set from the current position.

The "current position" and how it changes as the iterator is used to traverse the set of defined nodes OR the set of defined nodes changes needs to be defined more rigorously. The "current" position is not ON a node, but at a spot in the set just previous to the node that getNextNode() will return. The initial position of the iterator is just before the first node, so the first call to getNextNode() will return the first node; if getPreviousNode() is called at the initial position, *null* is returned.. If a node is inserted before the node just after the iterator position, it will be returned by getNextNode(); likewise if a node is inserted after the node just previous to the iterator position, it will be return by the next getPrevNode() call.

IDL Definition

```
interface NodeIterator {
  unsigned long       getLength();
  unsigned long       getCurrentPos();
  boolean             atFirst();
  boolean             atLast();
  Node                toNextNode();
  Node                toPrevNode();
  Node                toFirstNode();
  Node                toLastNode();
  Node                moveTo(in int n);
};
```

Method *`getLength()`*

This method returns the number of items that will be iterated over if the iterator is started at the beginning, and `toNext()` is called repeatedly until the end of the sequence is encountered. ***Note:*** *This may be an expensive operation.*

Parameters

This method has no parameters.

Return Values

unsigned long	This method will always return the correct number of items in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method *`getCurrentPos()`*

This method returns the current position of the iterator, where "0" is the initial position before the first node in the set. ***Note:*** *This may be an expensive operation.*

Parameters

This method has no parameters.

Return Values

unsigned long	This method will always return the correct position in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method `atFirst()`

Returns TRUE if the iterator is positioned at the beginning of the set of nodes it will iterate over.

Parameters

This method has no parameters.

Return Values

boolean	This method will always return the correct value in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method `atLast()`

Returns TRUE if the iterator is positioned at the end of the set of nodes it will iterate over.

Parameters

This method has no parameters.

Return Values

boolean	This method will always return the correct value in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method *`toNextNode()`*

This method returns the next node after the "current" position and resets the current position to be after the node returned.

Parameters

This method has no parameters.

Return Values

Node	This method returns the node that has been traversed to, or `null` when it is not possible to iterate any further.

Exceptions

This method throws no exceptions.

Method *`toPrevNode()`*

This method returns the node before the "current" position and resets the current position to be before the node returned.

Parameters

This method has no parameters.

Return Values

Node	This method returns the node that has been traversed to, or `null` when it is not possible to iterate any further.

Exceptions

This method throws no exceptions.

Method `toFirstNode()`

This method positions the iterator so that the current position is just before the first node in the set being iterated over.

Parameters

This method has no parameters.

Return Values

Node	Returns the first node in the set. This method will only return `null` when there are no items to iterate over.

Exceptions

This method throws no exceptions.

Method `toLastNode()`

This method positions the iterator so that the current position is after the last node in the set being iterated over.

Parameters

This method has no parameters.

Return Values

Node	Returns the last node in the set. This method will only return `null` when there are no items to iterate over.

Exceptions

This method throws no exceptions.

Method `moveTo()`

This method alters the internal state of the iterator such that the current position is before the *Nth* node in the sequence the iterator is presenting relative to the current position. ***Note:*** *This may be a slow operation.*

Parameters

n	The index of the element in the set to position the iterator before.

Return Values

Node	Returns the nth node. This method will return `null` when position specified is out of the range of legal values.

Exceptions

This method throws no exceptions.

Interface *TreeIterator*

A `TreeIterator` is a `NodeIterator` that has additional methods that are specific to tree navigation.

IDL Definition

```
interface TreeIterator : NodeIterator {
  unsigned long       numChildren();
  unsigned long       numPreviousSiblings();
  unsigned long       numNextSiblings();
  Node                toParent();
  Node                toPreviousSibling();
  Node                toNextSibling();
  Node                toFirstChild();
  Node                toLastChild();
  Node                toNthChild(in int n)  interface NoSuchNodeException {};
;
};
```

Method *numChildren()*

This method returns the number of children that lie below the current node.

Parameters

This method has no parameters.

Return Values

unsigned long	This method will always return the correct number of items in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method *numPreviousSiblings()*

This method returns the number of siblings that lie previous to the current node.

Parameters

This method has no parameters.

Return Values

unsigned long	This method will always return the correct number of items in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method *`numNextSiblings()`*

This method returns the number of siblings that lie after to the current node.

Parameters

This method has no parameters.

Return Values

unsigned long	This method will always return the correct number of items in a single threaded environment, but may return misleading results in multithreaded, multiuser situations..

Exceptions

This method throws no exceptions.

Method *`toParent()`*

Move the iterator to the parent of the current node, if there is a parent.

Parameters

This method has no parameters.

Return Values

Node	This method will return `null` when there is no parent for the current node.

Exceptions

This method throws no exceptions.

Method `toPreviousSibling()`

Move the iterator to the previous sibling of the current node, if there is one.

Parameters

This method has no parameters.

Return Values

Node	This method will return `null` when there is no previous sibling for the current node.

Exceptions

This method throws no exceptions.

Method `toNextSibling()`

Move the iterator to the next sibling of the current node, if there is one.

Parameters

This method has no parameters.

Return Values

Node	This method will return `null` when there is no next sibling for the current node.

Exceptions

This method throws no exceptions.

Method *toFirstChild()*

Move the iterator to the first child of the current node, if there is one.

Parameters

This method has no parameters.

Return Values

Node	This method will return `null` when there is no children for the current node.

Exceptions

This method throws no exceptions.

Method *toLastChild()*

Move the iterator to the last child of the current node, if there is one.

Parameters

This method has no parameters.

Return Values

Node	This method will return `null` when there is no children for the current node.

Exceptions

This method throws no exceptions.

Method `toNthChild()`

This method alters the internal state of the iterator such that the node it references is the *Nth* child of the current node.

Parameters

n	The index of the child to position the iterator before.

Return Values

Node	This method will return `null` when position specified is out of the range of legal values.

Exceptions

NoSuchNodeException	Thrown if the value specified is greater than the number that would be returned from `numChildren()`.

Interface *Attribute*

The Attribute object represents an attribute in an Element object. Typically the allowable values for the attribute are defined in a document type definition.

The attribute's effective value is determined as follows: if this attribute has been explicitly assigned any value, that value is the attribute's effective value; otherwise, if there is a declaration for this attribute, and that declaration includes a default value, then that default value is the attribute's effective value; otherwise, the attribute has no effective value.) Note, in particular, that an effective value of the null string would be returned as a Text node instance whose `toString` method will return a zero length string (as will `toString` invoked directly on this Attribute instance). If the attribute has no effective value, then this method will return `null`. Note the `toString` method on the Attribute instance can also be used to retrieve the string version of the attribute's value(s).

NOTE - In XML, the value of an attribute is represented by the child nodes of an attribute node, since the value can be and arbitrarily complex list of entity references. In HTML, the value can only be a string; whether implementations must expose this as Node is under discussion.

IDL Definition

```
interface Attribute {
  wstring             getName();
  wstring             getValue();
  attribute boolean        specified;
  wstring             toString();
};
```

Method *getName()*

Returns the name of this attribute.

Parameters

This method has no parameters.

Return Values

wstring	

Exceptions

This method throws no exceptions.

Method *getValue()*

Returns the string value of this attribute.

Parameters

This method has no parameters.

Return Values

wstring	

Exceptions

This method throws no exceptions.

Attribute *specified*

If this attribute was explicitly given a value in the original document, this will be true; otherwise, it will be false.

Method *toString()*

Returns the value of the attribute as a string. Character and general entity references will have been replaced wit their values in the returned string.

Parameters

This method has no parameters.

Return Values

wstring	

Exceptions

This method throws no exceptions.

Interface *AttributeList*

AttributeList objects are used to represent collections of Attribute objects which can be accessed by name. The Attribute objects contained in a AttributeList may also be accessed by ordinal index. In most cases, AttributeList objects are created from Element objects.

IDL Definition

```
interface AttributeList {
  Attribute           getAttribute(in wstring attrName);
  Attribute           setAttribute(in Attribute attr);
  Attribute           remove(in wstring attrName)  interface NoSuchAttributeException {};
;
  Attribute           item(in unsigned long index)  interface NoSuchAttributeException {};
;
  unsigned long       getLength();
};
```

Method *getAttribute()*

Retrieve an Attribute instance from the list by its name. If it's not present, `null` is returned.

Parameters

attrName	

Return Values

Attribute	

Exceptions

This method throws no exceptions.

Method *setAttribute()*

Add a new attribute to the end of the list and associate it with the given name. If the name already exists, the previous Attribute object is replaced, and returned. If no object of the same name exists, `null` is returned, and the named Attribute is added to the end of the AttributeList object; that is, it is accessible via the `item` method using the index one less than the value returned by `getLength`.

Parameters

attr	

Return Values

Attribute	

Exceptions

This method throws no exceptions.

Method *remove()*

Removes the Attribute instance named name from the list and returns it. If the name provided does not exist, the `NoSuchAttributeException` is thrown.

Parameters

attrName	

Return Values

Attribute	

Exceptions

NoSuchAttributeException	

Method *item()*

Returns the (zero-based) indexth Attribute item in the collection. If index is greater than or equal to the number of nodes in the list, a `NoSuchAttributeException` is thrown.

Parameters

index	

Return Values

Attribute	

Exceptions

NoSuchAttributeException	

Method *getLength()*

Returns the number of Attributes in the AttributeList instance.

Parameters

This method has no parameters.

Return Values

unsigned long	

Exceptions

This method throws no exceptions.

Interface *Element*

By far the vast majority (apart from text) of node types that authors will generally encounter when traversing a document will be Element nodes. These objects represent both the element itself, as well as any contained nodes. For example (in XML):

```
<elementExample> id="demo">
  <subelement1/>
  <subelement2><subsubelement/></subelement2>
</elementExample>
```

When represented using DOM, the top node would be "elementExample", which contains two child Element nodes (and some space), one for "subelement1" and one for "subelement2". "subelement1" contains no child nodes of its own.

IDL Definition

```
interface Element : Node {
  wstring             getTagName();
  NodeIterator        getAttributes();
  wstring             getAttribute(in name name);
  void                setAttribute(in string name,
                                   in string value);
  void                removeAttribute(in wstring name);
  Attribute           getAttributeNode(in name name);
  void                setAttributeNode(in Attribute newAttr);
  void                removeAttributeNode(in Attribute oldAttr);
  void                getElementsByTagName(in wstring tagname);
  void                normalize();
};
```

Method *getTagName()*

This method returns the string that is the element's name. For example, in:

```
<elementExample id="demo">
        ...
</elementExample>
```

This would have the value `"elementExample"`. Note that this is case-preserving, as are all of the operations of the DOM. See Name case in the DOM for a description of why the DOM preserves case.

Parameters

This method has no parameters.

Return Values

wstring	

Exceptions

This method throws no exceptions.

Method *getAttributes()*

The attributes for this element. In the `elementExample` example above, the attributes list would consist of the `id` attribute, as well as any attributes which were defined by the document type definition for this element which have default values.

Parameters

This method has no parameters.

Return Values

NodeIterator	

Exceptions

This method throws no exceptions.

Method *getAttribute()*

Retrieves an Attribute value by name from an Element. object.

Parameters

name	The name of the attribute to retrieve

Return Values

wstring	

Exceptions

This method throws no exceptions.

Method *setAttribute()*

Adds a new attribute/value pair to an Element node object. If an attribute by that name is already present in the element, it’s value is changed to be that of the value parameter.

Parameters

name	
value	

Return Values

void	

Exceptions

This method throws no exceptions.

Method *removeAttribute()*

Removes the specified attribute from an Element node object.

Parameters

name	

Return Values

void	

Exceptions

This method throws no exceptions.

Method `getAttributeNode()`

Retrieves an Attribute node by name from an Element. object.

Parameters

name	The name of the attribute to retrieve

Return Values

Attribute	

Exceptions

This method throws no exceptions.

Method `setAttributeNode()`

Adds a new attribute/value pair to an Element node object. If an attribute by that name is already present in the element, it's value is changed to be that of the Attribute instance.

Parameters

newAttr	

Return Values

void	

Exceptions

This method throws no exceptions.

Method *removeAttributeNode()*

Removes the specified attribute/value pair from an Element node object.

Parameters

oldAttr	

Return Values

void	

Exceptions

This method throws no exceptions.

Method *getElementsByTagName()*

Returns an iterator through all subordinate elements with a given tag name.

Parameters

tagname	The name of the tag to match on

Return Values

void	

Exceptions

This method throws no exceptions.

Method *normalize()*

Puts all Tet nodes in the sub-tree underneath this Element into a "normal" form where only markup (e.g., tags, comments, PIs, CDATASections, and and entity references separates Text nodes.

Parameters

This method has no parameters.

Return Values

void	

Exceptions

This method throws no exceptions.

Interface *Text*

The Text object contains the non-markup content of an Element. If there is no markup inside an Element's content, the text will be contained in a single Text object that is the child of the Element. Any markup will parse into child elements that are siblings of the Text nodes on either side of it, and whose content will be represented as Text node children of the markup element.

IDL Definition

```
interface Text : Node {
  attribute wstring        data;
  void                append(in wstring data);
  void                insert(in int offset,
                             in wstring data);
  void                delete(in int offset,
                             in int count);
  void                replace(in int offset,
                              in int count,
                              in wstring data);
  void                splice(in Element element,
                             in int offset,
                             in int count);
};
```

Attribute *data*

This holds the actual content of the text node. Text nodes contain just plain text, without markup and without entities, both of which are represented as separate objects in the DOM.

Method *append()*

Append the string to the end of the character data in this Text node

Parameters

data	String to append

Return Values

void	This method returns nothing.

Exceptions

This method throws no exceptions.

Method *insert()*

Insert the string at the specified character offset.

Parameters

offset	Offset at which to insert
data	String to insert

Return Values

void	This method returns nothing.

Exceptions

This method throws no exceptions.

Method `delete()`

Insert characters starting at the specified character offset.

Parameters

offset	Offset at which to delete
count	Number of characters to delete

Return Values

void	This method returns nothing.

Exceptions

This method throws no exceptions.

Method `replace()`

Replace the characters starting at the specified character offset with the specified string.

Parameters

offset	Offset at which to start replacing
count	Number of characters to replace
data	String to replace previous content with.

Return Values

void	This method returns nothing.

Exceptions

This method throws no exceptions.

Method *splice()*

Insert the specified element in the tree as a sibling of the Text node. The result of this operation may be the creation of up to 2 new Text nodes: the character data specified by the offset and count will form one Text node that will become the child of the inserted element and the remainder of the character data (after the offset and count) will form another Text node become a sibling of the original Text node.

Parameters

element	Element to insert into the tree.
offset	Offset at which to insert.
count	Number of characters to copy to child Text node of element.

Return Values

void	This method returns nothing.

Exceptions

This method throws no exceptions.

Interface *Comment*

Represents the content of a comment, i.e. all the characters between the starting '`<!--`' and *ending '`-->`'. Note that this is the definition of a comment in XML, and, in practice, HTML, although some HTML tools may implement the full SGML comment structure.

IDL Definition

```
interface Comment : Node {
  attribute wstring        data;
};
```

Attribute *`data`*

The content of the comment, exclusive of the comment begin and end sequence.

Interface *PI*

A PI node is a "processing instruction". The content of the PI node is the entire content between the delimiters of the processing instruction.

IDL Definition

```
interface PI : Node {
  attribute wstring        name;
  attribute wstring        data;
};
```

Attribute *`name`*

XML defines a name as the first token following the markup that begins the processing instruction, and this attribute returns that name. For HTML, the returned value is `null`.

Attribute *`data`*

The content of the processing instruction, from the character immediately after the `<?` (after the name in XML) to the character immediately preceding the `?>`.

2.9. IDL Interface definitions

This section contains the IDL definitions for the objects in the core Document Object Model. The HTML IDL definition is here [p.89] , and the XML IDL definition, including the types to represent the document type definition is here [p.65] .

2.9. IDL Interface definitions

```
interface DOM {
  Document            createDocument(in wstring type);
  boolean             hasFeature(in wstring feature);
};

interface DocumentContext {
  attribute Document       document;
};

interface DocumentFragment : Node {
  attribute Document       masterDoc;
};

interface Document : DocumentFragment {
  attribute Node           documentType;
  attribute Element        documentElement;
  attribute DocumentContextcontextInfo;
  DocumentContext     createDocumentContext();
  Element             createElement(in wstring tagName,
                                    in AttributeList attributes);
  Text                createTextNode(in wstring data);
  Comment             createComment(in wstring data);
  PI                  createPI(in wstring name,
                               in wstring data);
  Attribute           createAttribute(in wstring name,
                                      in Node value);
  AttributeList       createAttributeList();
  TreeIterator        createTreeIterator(in Node node);
  NodeIterator        getElementsByTagName(in wstring tagname);
};

interface Node {
  // NodeType
  const int            DOCUMENT             = 1;
  const int            ELEMENT              = 2;
  const int            ATTRIBUTE            = 3;
  const int            PI                   = 4;
  const int            COMMENT              = 5;
  const int            TEXT                 = 6;

  int                 getNodeType();
  Node                getParentNode();
  NodeIterator        getChildNodes();
  boolean             hasChildNodes();
  Node                getFirstChild();
  Node                getPreviousSibling();
  Node                getNextSibling();
  Node                insertBefore(in Node newChild,
                                   in Node refChild)  interface NotMyChildException {};
;
  Node                replaceChild(in Node newChild,
                                   in Node oldChild)  interface NotMyChildException {};
;
  Node                removeChild(in Node oldChild)  interface NotMyChildException {};
;
};

interface NodeIterator {
  unsigned long       getLength();
  unsigned long       getCurrentPos();
  boolean             atFirst();
  boolean             atLast();
  Node                toNextNode();
  Node                toPrevNode();
  Node                toFirstNode();
  Node                toLastNode();
  Node                moveTo(in int n);
};

interface TreeIterator : NodeIterator {
  unsigned long       numChildren();
  unsigned long       numPreviousSiblings();
  unsigned long       numNextSiblings();
  Node                toParent();
  Node                toPreviousSibling();
  Node                toNextSibling();
  Node                toFirstChild();
  Node                toLastChild();
  Node                toNthChild(in int n)  interface NoSuchNodeException {};
;
};

interface Attribute {
  wstring             getName();
  wstring             getValue();
  attribute boolean        specified;
  wstring             toString();
};

interface AttributeList {
  Attribute           getAttribute(in wstring attrName);
  Attribute           setAttribute(in Attribute attr);
  Attribute           remove(in wstring attrName)  interface NoSuchAttributeException {};
;
  Attribute           item(in unsigned long index)  interface NoSuchAttributeException {};
;
  unsigned long       getLength();
};

interface Element : Node {
  wstring             getTagName();
  NodeIterator        getAttributes();
  wstring             getAttribute(in name name);
  void                setAttribute(in string name,
                                   in string value);
  void                removeAttribute(in wstring name);
  Attribute           getAttributeNode(in name name);
  void                setAttributeNode(in Attribute newAttr);
  void                removeAttributeNode(in Attribute oldAttr);
  void                getElementsByTagName(in wstring tagname);
  void                normalize();
};

interface Text : Node {
  attribute wstring        data;
  void                append(in wstring data);
  void                insert(in int offset,
                             in wstring data);
  void                delete(in int offset,
                             in int count);
  void                replace(in int offset,
                              in int count,
                              in wstring data);
  void                splice(in Element element,
                             in int offset,
                             in int count);
};

interface Comment : Node {
  attribute wstring        data;
};

interface PI : Node {
  attribute wstring        name;
  attribute wstring        data;
};
```

2.10. Java Core API definitions

2.10. Java Core API definitions

```
public interface DOM {
   public Document          createDocument(String type);
   public boolean           hasFeature(String feature);
}

public interface DocumentContext {
   public Document          getDocument();
   public void              setDocument(Document arg);

}

public interface DocumentFragment extends Node {
   public Document          getMasterDoc();
   public void              setMasterDoc(Document arg);

}

public interface Document extends DocumentFragment {
   public Node              getDocumentType();
   public void              setDocumentType(Node arg);

   public Element           getDocumentElement();
   public void              setDocumentElement(Element arg);

   public DocumentContext   getContextInfo();
   public void              setContextInfo(DocumentContext arg);

   public DocumentContext   createDocumentContext();
   public Element           createElement(String tagName,
                                          AttributeList attributes);
   public Text              createTextNode(String data);
   public Comment           createComment(String data);
   public PI                createPI(String name,
                                     String data);
   public Attribute         createAttribute(String name,
                                            Node value);
   public AttributeList     createAttributeList();
   public TreeIterator      createTreeIterator(Node node);
   public NodeIterator      getElementsByTagName(String tagname);
}

public interface Node {
   // NodeType
   public static final int           DOCUMENT             = 1;
   public static final int           ELEMENT              = 2;
   public static final int           ATTRIBUTE            = 3;
   public static final int           PI                   = 4;
   public static final int           COMMENT              = 5;
   public static final int           TEXT                 = 6;

   public int               getNodeType();
   public Node              getParentNode();
   public NodeIterator      getChildNodes();
   public boolean           hasChildNodes();
   public Node              getFirstChild();
   public Node              getPreviousSibling();
   public Node              getNextSibling();
   public Node              insertBefore(Node newChild,
                                         Node refChild)  public class NotMyChildException extends Exception {};
;
   public Node              replaceChild(Node newChild,
                                         Node oldChild)  public class NotMyChildException extends Exception {};
;
   public Node              removeChild(Node oldChild)  public class NotMyChildException extends Exception {};
;
}

public interface NodeIterator {
   public int               getLength();
   public int               getCurrentPos();
   public boolean           atFirst();
   public boolean           atLast();
   public Node              toNextNode();
   public Node              toPrevNode();
   public Node              toFirstNode();
   public Node              toLastNode();
   public Node              moveTo(int n);
}

public interface TreeIterator extends NodeIterator {
   public int               numChildren();
   public int               numPreviousSiblings();
   public int               numNextSiblings();
   public Node              toParent();
   public Node              toPreviousSibling();
   public Node              toNextSibling();
   public Node              toFirstChild();
   public Node              toLastChild();
   public Node              toNthChild(int n)  public class NoSuchNodeException extends Exception {};
;
}

public interface Attribute {
   public String            getName();
   public String            getValue();
   public boolean           getSpecified();
   public void              setSpecified(boolean arg);

   public String            toString();
}

public interface AttributeList {
   public Attribute         getAttribute(String attrName);
   public Attribute         setAttribute(Attribute attr);
   public Attribute         remove(String attrName)  public class NoSuchAttributeException extends Exception {};
;
   public Attribute         item(int index)  public class NoSuchAttributeException extends Exception {};
;
   public int               getLength();
}

public interface Element extends Node {
   public String            getTagName();
   public NodeIterator      getAttributes();
   public String            getAttribute(name name);
   public void              setAttribute(String name,
                                         String value);
   public void              removeAttribute(String name);
   public Attribute         getAttributeNode(name name);
   public void              setAttributeNode(Attribute newAttr);
   public void              removeAttributeNode(Attribute oldAttr);
   public void              getElementsByTagName(String tagname);
   public void              normalize();
}

public interface Text extends Node {
   public String            getData();
   public void              setData(String arg);

   public void              append(String data);
   public void              insert(int offset,
                                   String data);
   public void              delete(int offset,
                                   int count);
   public void              replace(int offset,
                                    int count,
                                    String data);
   public void              splice(Element element,
                                   int offset,
                                   int count);
}

public interface Comment extends Node {
   public String            getData();
   public void              setData(String arg);

}

public interface PI extends Node {
   public String            getName();
   public void              setName(String arg);

   public String            getData();
   public void              setData(String arg);

}
```

2.11. ECMAScript Core API definitions

Object DOM

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

createDocument(type)
: This method returns a **Document**. The **type** parameter is expected to be of type **wstring.**.

hasFeature(feature)
: This method returns a **boolean**. The **feature** parameter is expected to be of type **wstring.**.

Object DocumentContext

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

document
: This property is expected to be a **Document**

Object DocumentFragment

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

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

masterDoc
: This property is expected to be a **Document**

Object Document

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

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

documentType
: This property is expected to be a **Node**

documentElement
: This property is expected to be a **Element**

contextInfo
: This property is expected to be a **DocumentContext**

createDocumentContext
: This property is expected to be a **DocumentContext**

createAttributeList
This property is expected to be a **AttributeList**

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

createElement(tagName, attributes)
This method returns a **Element**. The **tagName** parameter is expected to be of type **wstring.**. The **attributes** parameter is expected to be of type **AttributeList.**.
createTextNode(data)
This method returns a **Text**. The **data** parameter is expected to be of type **wstring.**.
createComment(data)
This method returns a **Comment**. The **data** parameter is expected to be of type **wstring.**.
createPI(name, data)
This method returns a **PI**. The **name** parameter is expected to be of type **wstring.**. The **data** parameter is expected to be of type **wstring.**.
createAttribute(name, value)
This method returns a **Attribute**. The **name** parameter is expected to be of type **wstring.**. The **value** parameter is expected to be of type **Node.**.
createTreeIterator(node)
This method returns a **TreeIterator**. The **node** parameter is expected to be of type **Node.**.
getElementsByTagName(tagname)
This method returns a **NodeIterator**. The **tagname** parameter is expected to be of type **wstring.**.

Object Node

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

nodeType
This property is expected to be a **int**
parentNode
This property is expected to be a **Node**
childNodes
This property is expected to be a **NodeIterator**
hasChildNodes
This property is expected to be a **boolean**
firstChild
This property is expected to be a **Node**
previousSibling
This property is expected to be a **Node**
nextSibling
This property is expected to be a **Node**

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

insertBefore(newChild, refChild)
This method returns a **Node**. The **newChild** parameter is expected to be of type **Node.**. The **refChild** parameter is expected to be of type **Node.**.

replaceChild(newChild, oldChild)
This method returns a **Node**. The **newChild** parameter is expected to be of type **Node.**. The **oldChild** parameter is expected to be of type **Node.**.

removeChild(oldChild)
This method returns a **Node**. The **oldChild** parameter is expected to be of type **Node.**.

Object NodeIterator

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

length
This property is expected to be a **unsigned long**

currentPos
This property is expected to be a **unsigned long**

atFirst
This property is expected to be a **boolean**

atLast
This property is expected to be a **boolean**

toNextNode
This property is expected to be a **Node**

toPrevNode
This property is expected to be a **Node**

toFirstNode
This property is expected to be a **Node**

toLastNode
This property is expected to be a **Node**

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

moveTo(n)
This method returns a **Node**. The **n** parameter is expected to be of type **int.**.

Object TreeIterator

TreeIterator has the all the properties and methods of **NodeIterator** as well as the properties and methods defined below.

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

numChildren
This property is expected to be a **unsigned long**

numPreviousSiblings
This property is expected to be a **unsigned long**

numNextSiblings
This property is expected to be a **unsigned long**
toParent
This property is expected to be a **Node**
toPreviousSibling
This property is expected to be a **Node**
toNextSibling
This property is expected to be a **Node**
toFirstChild
This property is expected to be a **Node**
toLastChild
This property is expected to be a **Node**

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

toNthChild(n)
This method returns a **Node**. The **n** parameter is expected to be of type **int.**.

Object Attribute

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

specified
This property is expected to be a **boolean**
name
This property is expected to be a **wstring**
value
This property is expected to be a **wstring**
toString
This property is expected to be a **wstring**

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

Object AttributeList

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

length
This property is expected to be a **unsigned long**

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

getAttribute(attrName)
This method returns a **Attribute**. The **attrName** parameter is expected to be of type **wstring.**.

setAttribute(attr)
This method returns a **Attribute**. The **attr** parameter is expected to be of type **Attribute.**.
remove(attrName)
This method returns a **Attribute**. The **attrName** parameter is expected to be of type **wstring.**.
item(index)
This method returns a **Attribute**. The **index** parameter is expected to be of type **unsigned long.**.

Object Element

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

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

tagName
This property is expected to be a **wstring**
attributes
This property is expected to be a **NodeIterator**
normalize
This property is expected to be a **void**

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

getAttribute(name)
This method returns a **wstring**. The **name** parameter is expected to be of type **name.**.
setAttribute(name, value)
This method returns a **void**. The **name** parameter is expected to be of type **string.**. The **value** parameter is expected to be of type **string.**.
removeAttribute(name)
This method returns a **void**. The **name** parameter is expected to be of type **wstring.**.
getAttributeNode(name)
This method returns a **Attribute**. The **name** parameter is expected to be of type **name.**.
setAttributeNode(newAttr)
This method returns a **void**. The **newAttr** parameter is expected to be of type **Attribute.**.
removeAttributeNode(oldAttr)
This method returns a **void**. The **oldAttr** parameter is expected to be of type **Attribute.**.
getElementsByTagName(tagname)
This method returns a **void**. The **tagname** parameter is expected to be of type **wstring.**.

Object Text

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

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

data
: This property is expected to be a **String**

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

append(data)
: This method returns a **void**. The **data** parameter is expected to be of type **wstring.**.

insert(offset, data)
: This method returns a **void**. The **offset** parameter is expected to be of type **int.**. The **data** parameter is expected to be of type **wstring.**.

delete(offset, count)
: This method returns a **void**. The **offset** parameter is expected to be of type **int.**. The **count** parameter is expected to be of type **int.**.

replace(offset, count, data)
: This method returns a **void**. The **offset** parameter is expected to be of type **int.**. The **count** parameter is expected to be of type **int.**. The **data** parameter is expected to be of type **wstring.**.

splice(element, offset, count)
: This method returns a **void**. The **element** parameter is expected to be of type **Element.**. The **offset** parameter is expected to be of type **int.**. The **count** parameter is expected to be of type **int.**.

Object Comment

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

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

data
: This property is expected to be a **String**

Object PI

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

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

name
: This property is expected to be a **String**

data
: This property is expected to be a **String**

3. Document Object Model (XML) Level 1

Editors

Mike Champion, *ArborText*
Gavin Nicol, *Inso EPS*

The DOM Level One (Core) specification defines a set of object definitions that are sufficient to represent a document instance (the objects that occur withing the document itself). This specification is extends the DOM Level One (Core) specification such that document type definitions, entities, CDATA marked sections can also be represented.

The objects and interfaces defined within this document are sufficient to allow validators and other applications that make use of a DTD (*Document Type Definition*) to be written. For editors, the interfaces defined here will probably be insufficient for fine-grained editing, where information about the *document type declaration* may be necessary, though structural isomorphism should be easily accomplished.

3.1. Introduction

The DOM Level One (Core) specification defines a set of object definitions that are sufficient to represent a document instance (the objects that occur within the document itself). This specification is extends the DOM Level One (Core) specification such that document type definitions, entities, CDATA marked sections can also be represented.

The objects and interfaces defined within this document are sufficient to allow validators and other applications that make use of a DTD (*Document Type Definition*) to be written. For editors, the interfaces defined here will probably be insufficient for fine-grained editing, where information about the *document type declaration* may be necessary, though structural isomorphism should be easily accomplished.

3.2. Descriptions of the basic XML objects

This section describes the XML specialization of the Node objects. used throughout the DOM.

Interface *XMLNode*

The XML implementation of the Node interface adds some methods that are needed to manipulate specific features of XML documents

IDL Definition

```
interface XMLNode {
  Node                getParentXMLNode(in boolean expandEntities);
  NodeIterator        getChildXMLNodes(in boolean expandEntities);
  boolean             hasChildXMLNodes(in boolean expandEntities);
  Node                getFirstXMLChild(in boolean expandEntities);
  Node                getPreviousXMLSibling(in boolean expandEntities);
  Node                getNextXMLSibling(in boolean expandEntities);
  EntityReference     getEntityReference();
  EntityDeclaration   getEntityDeclaration();
};
```

Method *getParentXMLNode()*

Returns the parent of the given Node instance. If this node is the root of the document object tree, or if this node is not part of a document tree, `null` is returned.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

Node	The parent of the node.

Exceptions

This method throws no exceptions.

Method *getChildXMLNodes()*

Returns a NodeIterator object that will enumerate all children of this node. If there are no children, an iterator that will return no nodes is returned. The content of the returned NodeIterator is "live" in the sense that changes to the children of the Node object that it was created from will be immediately reflected in the nodes returned by the iterator; it is not a static snapshot of the content of the Node. Similarly, changes made to the nodes returned by the iterator will be immediately reflected in the tree, including the set of children of the Node that the NodeIterator was created from.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

NodeIterator	An iterator through the children of the node.

Exceptions

This method throws no exceptions.

Method *`hasChildXMLNodes()`*

Returns `true` if the node has any children, `false` if the node has no children at all. This method exists both for convenience as well as to allow implementations to be able to bypass object allocation, which may be required for implementing `getChildNodes()`.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

boolean	True if the node has children.

Exceptions

This method throws no exceptions.

Method *getFirstXMLChild()*

Returns the first child of a node. If there is no such node, `null` is returned.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

Node	The first child of the node, or null.

Exceptions

This method throws no exceptions.

Method *getPreviousXMLSibling()*

Returns the node immediately preceding the current node in a breadth-first traversal of the tree. If there is no such node, `null` is returned.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

Node	The the node immediately preceeding, or null.

Exceptions

This method throws no exceptions.

Method *getNextXMLSibling()*

Returns the node immediately following the current node in a breadth-first traversal of the tree. If there is no such node, `null` is returned.

Parameters

expandEntities	TRUE if the view of the tree with parsed entities expanded should be navigated, FALSE if the view without parsed entities expanded should be navigated

Return Values

Node	The node immediately following, or null.

Exceptions

This method throws no exceptions.

Method *getEntityReference()*

When navigating XML trees with expandedEntities set to TRUE, the DOM programmer will on occasion get Nodes returned that are part of the expansion of an entity rather than actual nodes in the tree. This method returns the entity reference that generated a particular node, or NULL if it was not part of an entity reference expansion.

Parameters

This method has no parameters.

Return Values

EntityReference	The entity reference whose expansion yields a given node, or NULL if the node is not part of an entity expansion.

Exceptions

This method throws no exceptions.

Method *getEntityDeclaration()*

When navigating XML trees with expandedEntities set to TRUE, the DOM programmer will on occasion get Nodes returned that are part of the expansion of an entity rather than actual nodes in the tree. This method returns the declaration for the entity reference that generated a particular node, or NULL if it was not part of an entity reference expansion.

Parameters

This method has no parameters.

Return Values

EntityDeclaration	The entity declaration for a reference whose expansion yields a given node, or NULL if the node is not part of an entity expansion.

Exceptions

This method throws no exceptions.

3.3. Document Type Definition support overview

A Document Type Definition (DTD) defines three things:

1. A definition of a grammar for a markup language.
2. A (possibly empty) set of entity declarations.
3. A (possibly empty) set of notation declarations.

This specification gives access to all of these, though only in the post-parse form.

From a practical point of view, this means that while all the *information* contained within a DTD is available, not all of the information about what created it is. Parameter entity references, for example, are assumed to have been already expanded, and hence, their boundaries are lost.

3.4. Descriptions of objects related to the Document Type Definition

This section describes the objects that are used to represent the DTD of a document. The objects are not XML specific, though some attributes are specific to HTML DTD's. Such cases are clearly marked.

Interface *DocumentType*

Each document has a (possibly null) attribute that contains a reference to a `DocumentType` object. The `DocumentType` class provides an interface to access all of the entity declarations, notation declarations, and all the element type declarations.

IDL Definition

```
interface DocumentType {
  attribute wstring        name;
  attribute Node           externalSubset;
  attribute Node           internalSubset;
  attribute Node           generalEntities;
  attribute Node           parameterEntities;
  attribute Node           notations;
  attribute Node           elementTypes;
};
```

Attribute `name`

The `name` attribute is a `wstring` that holds the name of DTD; i.e. the name immediately following the `DOCTYPE` keyword.

Attribute `externalSubset`

The `externalSubset` attribute's children reference the list of nodes (definitions) that occurred in the external subset of a document. In this example:

```
<!DOCTYPE ex SYSTEM "ex.dtd" [
<ex/>
```

it would iterate over all of the declarations that occurred within the `ex.dtd` external entity. ***Note:*** *An iterator interface is used so as to not constrain implementations*

Attribute `internalSubset`

The internal subset's children constitute all the definitions that occurred within the internal subset of a document (the part that appears within the document instance). For example

```
<!DOCTYPE ex SYSTEM "ex.dtd" [
<!ENTITY ex "example">
]>
&lt;ex/>
```

if would iterate over a single node: the definition of the `ex` entity. ***Note:*** *An iterator interface is used so as to not constrain implementations*

Attribute ***generalEntities***

This is a `Node` whose children constitute the set of general entites that were defined within the external and the internal subset. For example in:

```
<!DOCTYPE ex SYSTEM "ex.dtd" [
<!ENTITY foo "foo">
<!ENTITY bar "bar">
<!ENTITY % baz "baz">
]>
<ex/>
```

the interface would provide access to `foo` and `bar` but not `baz`. All objects supporting the `Node` interface that are accessed though this attribute, will also support the `Entity` interface (defined below).

Attribute ***parameterEntities***

This is a `Node` whose children constitute the set of parameter entites that were defined within the external and the internal subset. In the example above, the interface would provide access to `baz` but not `foo` or `bar`. All objects supporting the `Node` interface that are accessed though this attribute, will also support the `Entity` interface (defined below).

Attribute ***notations***

This is a `Node` whose children constitute the set of notations that were defined within the external and the internal subset. All objects supporting the `Node` interface that are accessed though this attribute, will also support the `Notation` interface (defined below).

Attribute ***elementTypes***

This is a `Node` whose children constitute the set of element types that were defined within the external and the internal subset. All objects supporting the `Node` interface that are accessed though this attribute, will also support the `ElementDefinition` interface (defined below).

Interface *ElementDefinition*

The definition of each element defined within the external or internal subset (providing it is parsed), will be available through the `elementTypes` attribute of the `DocumentType` object. The name, attribute list, and content model are all available for inspection.

IDL Definition

```
interface ElementDefinition : Node {
  // ContentType
  const int           EMPTY               = 1;
  const int           ANY                 = 2;
  const int           PCDATA              = 3;
  const int           MODEL_GROUP         = 4;

  attribute wstring        name;
  attribute int            contentType;
  attribute ModelGroup     contentModel;
  attribute Node           attributeDefinitions;
  attribute Node           inclusions;
  attribute Node           exceptions;
};
```

Definition group *ContentType*

Defined Constants

EMPTY	The element is an empty element, and cannot have content.
ANY	The element may have character data, or any of the other elements defined within the DTD as content, in any order and sequence.
PCDATA	The element can have only PCDATA (*Parsed Character Data*) as content.
MODEL_GROUP	The element has a specific content model associated with it. The model is accessible through the `contentModel` attribute (below).

Attribute *name*

This is the name of the type of element being defined.

Attribute *contentType*

This attribute specifies the type of content of the element.

Attribute *contentModel*

If the `contentType` is `MODEL_GROUP`, then this will provide access to a `ModelGroup` (below) object that is the root of the content model object heirarchy for this element. For other content types, this will be null.

Attribute ***attributeDefinitions***

The children of this `Node` consist of the attributes that were defined to be on an `ElementDefinition`. Each object supporting the `Node` interface that is accessed through this attribute will also support the `AttributeDefinition` interface.

Attribute ***inclusions***

The children of this define a list of element type names that are included in the content model of this element by the SGML inclusion/exception mechanism (not available from XML, but used in HTML).

Attribute ***exceptions***

The children of this node define a list of element type names that are excluded from the content model of this element by the SGML inclusion/exception mechanism (not available from XML, but used in HTML).

Interface *PCDATAToken*

Token type for the string #PCDATA

IDL Definition

```
interface PCDATAToken : Node {
};
```

Interface *ElementToken*

Token for an element declaration.

IDL Definition

```
interface ElementToken : Node {
  // OccurrenceType
  const int           OPT                    = 1;
  const int           PLUS                   = 2;
  const int           REP                    = 3;

  attribute wstring        name;
  attribute int            occurrence;
};
```

Definition group *OccurrenceType*

Defined Constants

OPT	The `?` occurrence indicator.
PLUS	The + occurrence indicator.
REP	The `*` occurrence indicator.

Attribute *name*

The element type name.

Attribute *occurrence*

The number of times this element can occur.

Interface *ModelGroup*

The `ModelGroup` object represents the content model of an `ElementDefinition`. The content model is represented as a tree, where each node specifies how its children are connected, and the number of times that it can occur within its parent. Leaf nodes in the tree are either `PCDATAToken` or `ElementToken`.

IDL Definition

```
interface ModelGroup : Node {
  // OccurrenceType
  const int            OPT                    = 1;
  const int            PLUS                   = 2;
  const int            REP                    = 3;

  // ConnectionType
  const int            OR                     = 1;
  const int            SEQ                    = 2;
  const int            AND                    = 3;

  attribute int            occurrence;
  attribute int            connector;
  attribute Node           tokens;
};
```

Definition group *OccurrenceType*

Defined Constants

OPT	The `?` occurrence indicator.
PLUS	The `+` occurrence indicator.
REP	The `*` occurrence indicator.

Definition group *ConnectionType*

Defined Constants

OR	The `|` connection indicator.
SEQ	The `,` connection indicator.
AND	The `??` connection indicator.

Attribute *occurrence*

The number of times this model can occur.

Attribute *connector*

Describes how the `tokens` are connected together.

Attribute *tokens*

The children of this node define the list of tokens in this model group.

Interface *AttributeDefinition*

The `AttributeDefinition` interface is used to access information about a particular attribute definition on a given element. Object supporting this interface are available from the `ElementDefinition` object through the `attributeDefinitions` attribute.

IDL Definition

```
interface AttributeDefinition : Node {
  // DeclaredValueType
  const int           CDATA                 = 1;
  const int           ID                    = 2;
  const int           IDREF                 = 3;
  const int           IDREFS                = 4;
  const int           ENTITY                = 5;
  const int           ENTITIES              = 6;
  const int           NMTOKEN               = 7;
  const int           NMTOKENS              = 8;
  const int           NOTATION              = 9;
  const int           NAME_TOKEN_GROUP      = 10;

  // DefaultValueType
  const int           FIXED                 = 1;
  const int           REQUIRED              = 2;
  const int           IMPLIED               = 3;

  attribute wstring       name;
  attribute StringList    allowedTokens;
  attribute int           declaredType;
  attribute int           defaultType;
  attribute Node          defaultValue;
};
```

Definition group *DeclaredValueType*

Defined Constants

CDATA	
ID	
IDREF	
IDREFS	
ENTITY	
ENTITIES	
NMTOKEN	
NMTOKENS	
NOTATION	
NAME_TOKEN_GROUP	

Definition group *DefaultValueType*

Defined Constants

FIXED	
REQUIRED	
IMPLIED	

Attribute *name*

The name of the attribute.

Attribute *allowedTokens*

The list of tokens that are allowed as values. For example, in

```
&lt;!DOCTYPE ex [
&lt;!ELEMENT ex (#PCDATA) >
&lt;!ATTLIST ex test (FOO|BAR) "FOO" >
]>
&lt;ex>&lt;/ex>
```

this would hold `FOO` and `BAR`.

Attribute *declaredType*

This attribute indicates the type of values the attribute may contain.

Attribute *defaultType*

This specifies whether the attribute must be specified in the instance, and if it is not, what the attribute value will be if not provided.

Attribute *defaultValue*

This provides an interface to a `Node` whose children make up the default value for an attribute. This value is used if the attribute was not given an explicit value in the document instance.

Interface *Notation*

The `Notation` object is used to represent the definition of a notation within a DTD.

IDL Definition

```
interface Notation : Node {
  attribute wstring        name;
  attribute boolean        isPublic;
  attribute string         publicIdentifier;
  attribute string         systemIdentifier;
};
```

Attribute ***name***

This is the name of the notation.

Attribute ***isPublic***

If a public identifier was specified in the notation declaration, this will be `TRUE`, and the `publicIdentifier` attribute will contain the string for the public identifier.

Attribute ***publicIdentifier***

If a public identifier was specified in the notation declaration, this will hold the public identifier string, otherwise it will be null.

Attribute ***systemIdentifier***

If a system identifier was specified in the notation declaration, this will hold the system identifier string, otherwise it will be null.

3.5. Descriptions of objects related to Entities

XML contains the notion of "entities" that are declared in the DTD, defined in either internally in a document or in an external file, and referenced one or more places in the document. In any event, entities allow an abstract reference to some arbitrarily large or complex piece of text and markup.

Interface *EntityDeclaration*

.

IDL Definition

```
interface EntityDeclaration {
  attribute wstring        replacementString;
  attribute DocumentFragmentreplacementSubtree;
};
```

Attribute *replacementString*

The string that a reference to this entity is replaced with. It may contain markup and entity references. It does not apply to un-parsed entities.

Attribute *replacementSubtree*

The parsed subtree that references to this entity would logically point to. All markup in the replacement string is represented as sub-trees, and entity references are expanded.

Interface *EntityReference*

EntityReference objects are inserted into the initial structure model by the XML processor. XML does not mandate that a non-validating XML processor read and process entity declarations made in the external subset or that are declared in external parameter entities. This means that parsed entitiesthat are declared in the external subset need not be expanded by some classes of applications.

XML contains the notion of " parsed entities" that are declared in the DTD, defined in either internally in a document or in an external file, and referenced one or more places in the document. In any event, parsed entities allow an abstract reference to some arbitrarily large or complex piece of text and markup. Parsed entities are one type of general entity. The other type of general entity that XML defines is the unparsed entity, used for embedding data that is not in XML format in an XML document. it is commonly used for images.

Parameter entities are used in DTDs for similar purposes as parsed entities in document instances; namely they allow an abstract reference to some part of the DTD. XML defines a number of rules about the allowed content of a parameter entity; the reader is referred to the XML specification for details.

IDL Definition

```
interface EntityReference {
  attribute boolean        isExpanded;
  void                expand(in  );
};
```

Attribute *isExpanded*

The default view of entities is to be expanded.

Method *expand()*

Parameters

Return Values

void	

Exceptions

This method throws no exceptions.

3.6. Descriptions of additional objects in instances

XML adds to the range of objects that are available in document instances. These objects defined below.

Interface *CDATASection*

CDATA sections are used in the document instance, and provide a region in which most of the XML delimiter recognition does not take place. The primary purpose is for including material such as XML fragments, without needing to escape all the delimiters.

The *wstring* attribute of the Text node holds the text that was contained by the CDATA section. Note that this *may* contain characters that need to be escaped outside of CDATA sections.

IDL Definition

```
interface CDATASection : Text {
};
```

3.7. IDL Interface Definitions

```
interface XMLNode {
  Node                getParentXMLNode(in boolean expandEntities);
  NodeIterator        getChildXMLNodes(in boolean expandEntities);
  boolean             hasChildXMLNodes(in boolean expandEntities);
  Node                getFirstXMLChild(in boolean expandEntities);
  Node                getPreviousXMLSibling(in boolean expandEntities);
  Node                getNextXMLSibling(in boolean expandEntities);
  EntityReference     getEntityReference();
  EntityDeclaration   getEntityDeclaration();
};

interface DocumentType {
  attribute wstring        name;
  attribute Node           externalSubset;
  attribute Node           internalSubset;
  attribute Node           generalEntities;
  attribute Node           parameterEntities;
  attribute Node           notations;
  attribute Node           elementTypes;
};

interface ElementDefinition : Node {
  // ContentType
  const int            EMPTY                = 1;
  const int            ANY                  = 2;
  const int            PCDATA               = 3;
  const int            MODEL_GROUP          = 4;

  attribute wstring        name;
  attribute int            contentType;
  attribute ModelGroup     contentModel;
  attribute Node           attributeDefinitions;
  attribute Node           inclusions;
  attribute Node           exceptions;
};

interface PCDATAToken : Node {
};

interface ElementToken : Node {
  // OccurrenceType
  const int            OPT                  = 1;
  const int            PLUS                 = 2;
  const int            REP                  = 3;

  attribute wstring        name;
  attribute int            occurrence;
};

interface ModelGroup : Node {
  // OccurrenceType
  const int            OPT                  = 1;
  const int            PLUS                 = 2;
  const int            REP                  = 3;

  // ConnectionType
  const int            OR                   = 1;
  const int            SEQ                  = 2;
  const int            AND                  = 3;

  attribute int            occurrence;
  attribute int            connector;
  attribute Node           tokens;
};

interface AttributeDefinition : Node {
  // DeclaredValueType
  const int            CDATA                = 1;
  const int            ID                   = 2;
  const int            IDREF                = 3;
  const int            IDREFS               = 4;
  const int            ENTITY               = 5;
  const int            ENTITIES             = 6;
  const int            NMTOKEN              = 7;
  const int            NMTOKENS             = 8;
  const int            NOTATION             = 9;
  const int            NAME_TOKEN_GROUP     = 10;

  // DefaultValueType
  const int            FIXED                = 1;
  const int            REQUIRED             = 2;
  const int            IMPLIED              = 3;

  attribute wstring        name;
  attribute StringList     allowedTokens;
  attribute int            declaredType;
  attribute int            defaultType;
  attribute Node           defaultValue;
};

interface Notation : Node {
  attribute wstring        name;
  attribute boolean        isPublic;
  attribute string         publicIdentifier;
  attribute string         systemIdentifier;
};

interface EntityDeclaration {
  attribute wstring        replacementString;
  attribute DocumentFragmentreplacementSubtree;
};

interface EntityReference {
  attribute boolean        isExpanded;
  void                expand(in  );
};

interface CDATASection : Text {
};
```

3.8. Java XML API definitions

3.8. Java XML API definitions

```java
public interface XMLNode {
   public Node               getParentXMLNode(boolean expandEntities);
   public NodeIterator       getChildXMLNodes(boolean expandEntities);
   public boolean            hasChildXMLNodes(boolean expandEntities);
   public Node               getFirstXMLChild(boolean expandEntities);
   public Node               getPreviousXMLSibling(boolean expandEntities);
   public Node               getNextXMLSibling(boolean expandEntities);
   public EntityReference    getEntityReference();
   public EntityDeclaration  getEntityDeclaration();
}

public interface DocumentType {
   public String             getName();
   public void               setName(String arg);

   public Node               getExternalSubset();
   public void               setExternalSubset(Node arg);

   public Node               getInternalSubset();
   public void               setInternalSubset(Node arg);

   public Node               getGeneralEntities();
   public void               setGeneralEntities(Node arg);

   public Node               getParameterEntities();
   public void               setParameterEntities(Node arg);

   public Node               getNotations();
   public void               setNotations(Node arg);

   public Node               getElementTypes();
   public void               setElementTypes(Node arg);

}

public interface ElementDefinition extends Node {
   // ContentType
   public static final int            EMPTY                = 1;
   public static final int            ANY                  = 2;
   public static final int            PCDATA               = 3;
   public static final int            MODEL_GROUP          = 4;

   public String             getName();
   public void               setName(String arg);

   public int                getContentType();
   public void               setContentType(int arg);

   public ModelGroup         getContentModel();
   public void               setContentModel(ModelGroup arg);

   public Node               getAttributeDefinitions();
   public void               setAttributeDefinitions(Node arg);

   public Node               getInclusions();
   public void               setInclusions(Node arg);

   public Node               getExceptions();
   public void               setExceptions(Node arg);

}

public interface PCDATAToken extends Node {
}

public interface ElementToken extends Node {
   // OccurrenceType
   public static final int            OPT                  = 1;
   public static final int            PLUS                 = 2;
   public static final int            REP                  = 3;

   public String             getName();
   public void               setName(String arg);

   public int                getOccurrence();
   public void               setOccurrence(int arg);

}

public interface ModelGroup extends Node {
   // OccurrenceType
   public static final int            OPT                  = 1;
   public static final int            PLUS                 = 2;
   public static final int            REP                  = 3;

   // ConnectionType
   public static final int            OR                   = 1;
   public static final int            SEQ                  = 2;
   public static final int            AND                  = 3;

   public int                getOccurrence();
   public void               setOccurrence(int arg);

   public int                getConnector();
   public void               setConnector(int arg);

   public Node               getTokens();
   public void               setTokens(Node arg);

}

public interface AttributeDefinition extends Node {
   // DeclaredValueType
   public static final int            CDATA                = 1;
   public static final int            ID                   = 2;
   public static final int            IDREF                = 3;
   public static final int            IDREFS               = 4;
   public static final int            ENTITY               = 5;
   public static final int            ENTITIES             = 6;
   public static final int            NMTOKEN              = 7;
   public static final int            NMTOKENS             = 8;
   public static final int            NOTATION             = 9;
   public static final int            NAME_TOKEN_GROUP     = 10;

   // DefaultValueType
   public static final int            FIXED                = 1;
   public static final int            REQUIRED             = 2;
   public static final int            IMPLIED              = 3;

   public String             getName();
   public void               setName(String arg);

   public String             getAllowedTokens();
   public void               setAllowedTokens(String arg);

   public int                getDeclaredType();
   public void               setDeclaredType(int arg);

   public int                getDefaultType();
   public void               setDefaultType(int arg);

   public Node               getDefaultValue();
   public void               setDefaultValue(Node arg);

}

public interface Notation extends Node {
   public String             getName();
   public void               setName(String arg);

   public boolean            getIsPublic();
   public void               setIsPublic(boolean arg);

   public String             getPublicIdentifier();
   public void               setPublicIdentifier(String arg);

   public String             getSystemIdentifier();
   public void               setSystemIdentifier(String arg);

}

public interface EntityDeclaration {
   public String             getReplacementString();
   public void               setReplacementString(String arg);

   public DocumentFragment   getReplacementSubtree();
   public void               setReplacementSubtree(DocumentFragment arg);

}

public interface EntityReference {
   public boolean            getIsExpanded();
   public void               setIsExpanded(boolean arg);

   public void               expand( );
}

public interface CDATASection extends Text {
}
```

3.9. ECMAScript XML API definitions

Object XMLNode

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

entityReference
: This property is expected to be a **EntityReference**

entityDeclaration
: This property is expected to be a **EntityDeclaration**

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

getParentXMLNode(expandEntities)
: This method returns a **Node**. The **expandEntities** parameter is expected to be of type **boolean.**.

getChildXMLNodes(expandEntities)
: This method returns a **NodeIterator**. The **expandEntities** parameter is expected to be of type **boolean.**.

hasChildXMLNodes(expandEntities)
: This method returns a **boolean**. The **expandEntities** parameter is expected to be of type **boolean.**.

getFirstXMLChild(expandEntities)
: This method returns a **Node**. The **expandEntities** parameter is expected to be of type **boolean.**.

getPreviousXMLSibling(expandEntities)
: This method returns a **Node**. The **expandEntities** parameter is expected to be of type **boolean.**.

getNextXMLSibling(expandEntities)
: This method returns a **Node**. The **expandEntities** parameter is expected to be of type **boolean.**.

Object DocumentType

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

name
: This property is expected to be a **String**

externalSubset
: This property is expected to be a **Node**

internalSubset
: This property is expected to be a **Node**

generalEntities
: This property is expected to be a **Node**

parameterEntities
This property is expected to be a **Node**
notations
This property is expected to be a **Node**
elementTypes
This property is expected to be a **Node**

Object ElementDefinition

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

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

name
This property is expected to be a **String**
contentType
This property is expected to be a **int**
contentModel
This property is expected to be a **ModelGroup**
attributeDefinitions
This property is expected to be a **Node**
inclusions
This property is expected to be a **Node**
exceptions
This property is expected to be a **Node**

Object PCDATAToken

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

Object ElementToken

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

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

name
This property is expected to be a **String**
occurrence
This property is expected to be a **int**

Object ModelGroup

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

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

occurrence
: This property is expected to be a **int**

connector
: This property is expected to be a **int**

tokens
: This property is expected to be a **Node**

Object AttributeDefinition

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

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

name
: This property is expected to be a **String**

allowedTokens
: This property is expected to be a **String**

declaredType
: This property is expected to be a **int**

defaultType
: This property is expected to be a **int**

defaultValue
: This property is expected to be a **Node**

Object Notation

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

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

name
: This property is expected to be a **String**

isPublic
: This property is expected to be a **boolean**

publicIdentifier
: This property is expected to be a **String**

systemIdentifier
This property is expected to be a **String**

Object EntityDeclaration

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

replacementString
This property is expected to be a **String**
replacementSubtree
This property is expected to be a **DocumentFragment**

Object EntityReference

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

isExpanded
This property is expected to be a **boolean**

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

expand()
This method returns a **void**. The parameter is expected to be of type **.**.

Object CDATASection

CDATASection has the all the properties and methods of **Text** as well as the properties and methods defined below.

4. Document Object Model (HTML) Level 1

Editors:
Mike Champion, ArborText
Vidur Apparao, Netscape
Scott Isaacs, Microsoft (until January 1998)
Chris Wilson, Microsoft (after January 1998)

The Document Object Model (DOM) level one provides a mechanism for software developers and web script authors to access and manipulate parsed HTML and XML content. This document defines a set of objects that extends the Document Object Model (Core) such that the combination can represent all parts of a parsed HTML document. This document is meant to be concurrent with the HTML version 4.0 specification.

Introduction

This specification is part of the W3C Document Object Model [p.1] , which defines a programming interface for manipulating HTML and XML documents. Objects and methods which are used for both HTML and XML documents are defined in the W3C DOM Level One Core specification, which forms the basis for the W3C DOM Level One HTML specification that you are now reading. This document extends the Level One Core specification to describe objects and methods specific to HTML documents. In general, the functionality needed to manipulate hierarchical document structures, elements, and attributes will be found in the core specification; functionality that depends on the specific elements defined in HTML will be found in this document. Another document, the W3C DOM Level One XML specification, contains methods and objects specific to XML documents.

The goals of the HTML-specific DOM specification are:

- to specialize and add functionality that relates specifically to HTML documents and elements.
- to address issues of backwards compatibility with the Level 0 Document Object Model.
- to provide convenience mechanisms where appropriate for common and frequent operations on HTML documents.

The key differences between the core DOM and the HTML application of DOM is that the HTML Document Object Model exposes a number of convenience methods and properties that are consistent with the existing models and are more appropriate to script writers. In many cases, these enhancements are not applicable to a general DOM because they rely on the presence of a predefined DTD. For DOM Level 1, the DTD for HTML 4.0 is assumed.

The W3C DOM Working Group attempts, wherever possible, to maintain compatibility with the DOM Level 0, which is defined as being the functionality exposed in Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0.

More specifically, this document includes the following specializations for HTML:

- An HTMLDocument interface, derived from the core Document interface. HTMLDocument specifies the operations and queries that can be made on a HTML document.
- An HTMLElement interface, derived from the core Element interface. HTMLElement specifies the operations and queries that can be made on any HTML element. Methods on HTMLElement include those that allow for the retrieval and modification of attributes that apply to all HTML element.
- Specializations for all HTML elements that have attributes that extend beyond those specified in the HTMLElement interface. For all such attributes, the derived interface for the element contains explicit methods for setting and getting the values.

The Level 1 document does not include mechanisms to access and modify style specified through CSS-1. Furthermore, it will not define an event model for HTML documents. This functionality will be specified in a future Level of this specification. The events that are supported in Level 0 are given in the DOM Level 1 HTML ECMAScript binding for completeness; the method of defining and handling events is expected to change.

HTML Application of Core DOM

Naming Conventions

The HTML DOM follows a naming convention for properties, methods, events, collections, and data types. All names are defined as one or more English words concatenated together to form a single string.

Properties and Methods

The property or method name starts with the initial keyword in lowercase, and each subsequent word starts with a capital letter. For example, a property that returns document meta information such as the date the file was created might be named "fileDateCreated". In the ECMAScript binding, properties are exposed as properties of a given object. In Java, properties are exposed with get and set methods.

HTML Document

HTMLDocument

The HTMLDocument is the root of the HTML hierarchy and contains the entire content as its descendants.

Properties

`NamedNodeList images`
: A collection of all the images in the document

`NamedNodeList applets`
: A collection of all the <APPLET> (deprecated element) and <OBJECT> elements in the document.

`NamedNodeList links`
: A collection of all the <AREA> and <A HREF=...> elements in the document.

`NamedNodeList forms`
: A collection of all the forms in the document

`NamedNodeList anchors`
: A collection of all the <A NAME=...> elements in the document.

`wstring title`
: The title of a document as specified by the <TITLE> tag in the head of the document.

`wstring referrer`
: Returns the URL of the page that linked to this page. This property is an empty string if the user navigated to the page directly (not through a link, eg., from a bookmark).

`wstring cookie`
: Assigning a cookie string is in the following format: name=value; expires=date. Retrieving the cookie a semi-colon delimited list of name, value pairs for all the cookies associated with the page.

`wstring fileSize`
: The size of the document

`wstring fileCreatedDate`
: A string representation of the date the document was created.

`wstring fileModifiedDate`

`wstring fileUpdatedDate`
: The last update timestamp.

`wstring domain`
: The domain name of the server that served the document.

`wstring URL`
: The complete URL of the document.

`HTMLElement body`
: The element that contains the content for the document. In documents with BODY contents, returns the BODY element, and in frameset documents, this returns the outermost FRAMESET element.

The following properties have been deprecated in favor of the corresponding ones for the BODY element:

```
alinkColor
background
bgColor
fgColor
linkColor
vlinkColor
```

Methods

The following methods allow a user to add to or replace the structure model of a document using strings of unparsed HTML. The Working Group is currently considering alternate methods for providing similar functionality for both HTML and XML documents. The following methods may be deprecated at some point in the future in favor of a more general-purpose mechanism.

`void open();`
: Open a document stream for writing.

`void write(in wstring text);`
: Write a string of text to a document. The text is parsed into the document's structure model.

`void writeln(in wstring text);`
: Write a string of text followed by a newline character to a document. The text is parsed into the document's structure model.

`void close();`
: Closes a document stream and forces display.

LOCATION Object

The location object parses the URL for the current page into a number of properties providing easy access. The Location object is included in the HTML-specific interface ("window") in Level 0. In a subsequent level of the DOM, the DocumentContext object will be defined as part of the core API and the Location object and its methods may be deprecated.

HTML Elements

```
Element

  |

  +--- HTMLElement

          |

          +--- Specific HTML

               Elements
```

HTMLElement Object Hierarchy

Property Attributes

All HTML attributes are exposed as properties on the element object. The name of the exposed property always uses the naming conventions, and is independent of the case of the attribute in the source document. The data type of the property is determined by the type of the attribute as determined by the HTML 4.0 DTD. The attributes have the semantics given in the HTML 4.0 specification.

The attributes are exposed as properties for compatibility with Level 0. This usage is deprecated because it can not be generalised to all possible attribute names, as is required both for XML and potentially for future versions of HTML. We are considering how best to add, get, and remove attributes in a simple and direct way.

DTD Data Type	Object Model Data Type
CDATA	String
Value list (e.g.. (left | right | center))	String
one-value Value list (e.g. (border))	Boolean
Number	Long Integer

The return value of an attribute that has data type that is a value list is always capitalized, independent of the case of the value in the source document. For example, if the value of the align attribute on a P element is "left" then it is returned as "Left". For attributes with the CDATA data type, the value is returned with the capitalization as in the source document.

Naming Exceptions

To avoid name-space conflicts, an attribute with the same name as a keyword in one of our chosen binding languages will be prefixed. For HTML, the prefix used is "html" and for CSS, the prefix used is "css". For example, the "for" attribute collides with loop construct naming conventions and is named "htmlFor", while the CSS float attribute collides with the float data type and is named "cssFloat".

Exposing Element Type Names (tagName)

The property that exposes the element type name will return this in uppercase. For example, the body element type name is exposed through the "tagName" property as "BODY".

HTMLElement

All HTML elements derive from this class. Elements that only expose the core attributes are represented by the base HTMLElement class. These elements are as follows:

- HEAD
- special: SUB, SUP, SPAN, BDO
- font: TT, I, B, U, S, STRIKE, BIG, SMALL
- phrase: EM, STRONG, DFN, CODE, SAMP, KBD, VAR, CITE, ACRONYM
- list: DD
- NOFRAMES, NOSCRIPT
- ADDRESS, CENTER, PRE

Properties

`wstring title`
: Advisory title.

`wstring lang`
: Language code.

`wstring dir`
: Direction for weak/neutral text.

`wstring className`
: the class attribute of the element. This attribute has been renamed due to conflicts with the class keyword exposed by many languages.

`HTMLStyle style`
: RESERVED - For accessing the inline style of the element

Head Elements

LINK Element

The LINK element specifies a link to an external resource, and defines this document's relationship to that resource (or vice versa).

Properties

`wstring href`
: The base URL

`wstring charset`
: The character set encoding of the linked resource

`wstring hreflang`
: Language code

`wstring target`
: Render in this frame

`wstring rel`
: Forward link type

`wstring rev`
: Reverse link type

`wstring type`
: Advisory content type

`wstring media`
: Designed for use with specific media.

`bool disabled`
: Enables/disables the link. This is currently only used for stylesheet links, and may be used to activate or deactive stylesheets.

TITLE Element

The document title

Properties

`wstring title text`
: The specified title as a string

META Element

This contains generic meta-information about the document.

Properties

`wstring httpEquiv`
: HTTP response header name

`wstring content`
: Associated information

`wstring name`
: Meta-information name

`wstring scheme`
: Select form of content

BASE Element

Document base URL

Properties

`wstring href`
: The URL

`wstring target`
: The default target frame

ISINDEX Element

Represents the deprecated ISINDEX element in HTML (used for single-line text input).

Properties

`wstring prompt`
: Prompt message

STYLE Element

Style information - a more detailed stylesheet object model will be defined in a separate document.

Properties

`wstring type`
: The style sheet language (Internet media type).

`bool disabled`
: Enables/disables the stylesheet.

`wstring media`
: The intended destination medium for the style sheet.

BODY Element

The HTML document body. This element is always present in the DOM API, even if the tags are not present in the source document.

Properties

`wstring background`
: URL of the background texture tile image

`wstring bgColor`
: Background color code

`wstring text`
: Document text color

`wstring link`
: Color of links

`wstring vLink`
: Color of visited links

`wstring aLink`
: Color of active links

Form Elements

If any of the form control elements is not contained within a form, the form property returns null.

FORM Element

The form element encompasses behavior similar to a collection and an element. It provides direct access to the contained input elements as well as the attributes of the form element.

Properties

`wstring action`
: Server-side form handler

`wstring method`
: HTTP method used to submit form

`wstring target`
    Render in this frame
`wstring name`
    Name of frame for targeting
`long length`
    Length (read-only)
`wstring enctype`
`wstring acceptCharset`
    List of supported character sets
`NamedNodeList elements`
    Returns a collection of all elements in the form

SELECT Element

The select element allows the selection of an option. The contained options can be directly accessed through the select element as a collection.

Properties

`wstring type`
    The type of control created
`long size`
    Number of rows visible
`boolean multiple`
    May multiple items be selected?
`wstring name`
    Field name
`NamedNodeList options`
    The list of OPTION elements contained by this element.
`wstring type`
    What kind of select element
`wstring value`
    Property value
`boolean disabled`
    Control unavailable in this context?
`long length`
    Number of characters
`long tabIndex`
    Position in tabbing order
`long selectedIndex`
    The ordinal index of the selected item. -1 is returned if no element is selected. For multi-select list boxes, the first selected item is returned.
`HTMLFormElement form`
    Returns the form the element is contained within. Returns null if not within the context of a form. (read-only)

Methods

`void blur()`
  Remove keyboard focus
`void focus()`
  Set keyboard focus
`void add(in HTMLElement element, in HTMLElement before);`
  Add a new element to the Select collection
`void remove(in long index)`
  Remove an element from the Select collection

OPTION Element

A selectable choice

Properties

`wstring label`
  For use in hierarchical menus
`boolean defaultSelected`
  Specifies the initial selection of the option (the value of the SELECTED attribute).
`boolean disabled`
  Control unavailable in this context?
`boolean selected`
  Specifies the current selection of the option.
`wstring value`
  The current value of the option element.
`wstring text`
  The text contained within the option element.
`HTMLFormElement form`
  Returns the form the element is contained within. Returns null if not within the context of a form. (read-only)
`long index`
  The index of this OPTION in its parent SELECT.

INPUT Element

Form control

Security Notes: Depending upon the environment the page is being viewed, the value property may be read-only for the file upload input type. For the password input type, the actual value returned may be masked to prevent unauthorized use.

Properties

`wstring value`
: Used for radios and checkboxes

`wstring name`
: Reflects the NAME attribute (for backwards compatibility).

`boolean disabled`
: Control unavailable in this context?

`wstring defaultValue`
: Value if not explicitly specified

`long size`
: Size information - specific to each type of field

`long maxLength`
: Maximum number of characters for text fields

`boolean readOnly`
: For text and password

`boolean checked`
: For radio buttons and check boxes - TRUE if explicitly set. Represents the current state of the checkbox or radio button.

`boolean defaultChecked`
: For radio buttons and check boxes. Represents the value of the CHECKED HTML attribute.

`wstring type`
: The type of control created (reflects the TYPE attribute).

`wstring src`
: When the type attribute has the value "image", this attribute specifies the location of the image to be used to decorate the graphical submit button.

`wstring alt`
: Alternate text for user agents that can’t display this element.

`wstring usemap`
: Used to associate an image map with an element.

`wstring align`
: The horizontal alignment of the element.

`long tabIndex`
: Index used for tab navigation.

`wstring accessKey`
: A single character access key for the form element.

`wstring accept`
: A comma-separated list of content types that a server processing this form will handle correctly

`HTMLFormElement form`
: Returns the form the element is contained within. Returns null if not within the context of a form. (read-only)

Methods

`void blur()`
    Remove keyboard focus
`void focus()`
    Set keyboard focus
`void select()`
    Select the contents of the textarea (for elements with a TYPE attribute of Text, File, Password)
`void click()`
    Simulate a mouse-click (for elements with a TYPE attribute of Button, Checkbox, Radio, Reset, Submit)

TEXTAREA Element

Multi-line text field

Properties

`wstring type`
    The type of this form control.
`wstring value`
    Property value
`wstring name`
    Property name
`boolean disabled`
    Control unavailable in this context?
`wstring defaultValue`
    Value if no input from user
`boolean readOnly`
    Can user change the value?
`long rows`
    Number of text rows
`long cols`
    Width in columns
`long tabIndex`
    Position in tabbing order
`HTMLFormElement form`
    Returns the form the element is contained within. Returns null if not within the context of a form. (read-only).

Methods

`void blur()`
    Remove keyboard focus

`void focus()`
: Set keyboard focus

`void select()`
: Select the contents of the textarea

BUTTON Element

Push button

Properties

`wstring type`
: The type of this form control.

`wstring value`
: Property value

`wstring name`
: Property name

`boolean disabled`
: Control unavailable in this context?

`long tabIndex`
: Position in tabbing order

`wstring accessKey`
: Accessibility key character

`HTMLFormElement form`
: Returns the form the element is contained within. Returns null if not within the context of a form. (read-only).

LABEL Element

Form field label text

`wstring htmlFor`
: The HTML "for" needs to be named htmlFor. For as a property name collides with the keywords of most languages.

`wstring accessKey`
: Accessibility key character

LEGEND Element

Fieldset legend

Properties

`wstring align`
: Text alignment relative to fieldset

List Elements

UL Elements

Unordered list.

Properties

`boolean compact`
: Reduce inter-item spacing.

`wstring type`
: Bullet style

OL Elements

Ordered list.

Properties

`boolean compact`
: Reduce inter-item spacing.

`wstring type`
: Bullet style

`long start`
: Starting sequence number (for OL)

DL, DIR, MENU Elements

Definition list, directory list, menu list

Properties

`boolean compact`
: Reduce inter-item spacing

LI Element

List item

Properties

`wstring type`
: List item style

`long value`
: Reset sequence number

Block Elements

DIV Element

Generic container

Properties

`wstring align`
: Text alignment

P Element

Paragraph

`wstring align`
: Text alignment

H1, H2, H3, H4, H5, H6 Elements

Headings

Properties

`wstring align`
: Text alignment

Q, BLOCKQUOTE Elements

Quoted text

Properties

`wstring cite`
: URL for source document or message

Phrase Elements

BASEFONT Element

Base font

Properties

`wstring color`
: Font color

`wstring face`
: Font face id

`long size`
: Font size

FONT Element

Local change to font

Properties

`wstring color`
: Font color

`wstring face`
: Font face

`long size`
: Font size

INS & DEL Elements

Inserted or deleted text

Properties

`wstring cite`
: Reason for change

`wstring dateTime`
: Date and time of change

A Element

Anchor

Properties

`wstring target`
: Render in this frame

`wstring href`
: URL for linked resource

`wstring rel`
: Forward link type

`wstring rev`
: Reverse link type

`wstring name`
: Property name

`wstring accessKey`
: Accessibility key character

`long tabIndex`
: Position in tabbing order

`wstring charset`
: Character encoding of linked resource

Methods

`void blur()`
: Remove keyboard focus

`void focus()`
: Set keyboard focus

Replaced Elements

IMG Element

Embedded image

Properties

`boolean isMap`
: Use server-side image map

`wstring useMap`
: Use client side image map

`wstring border`
: Link border width

`long vspace`
: Vertical gutter

`long hspace`
: Horizontal gutter

`wstring alt`
: Short description

`wstring src`
: Source URL

`wstring align`
: Image alignment

`long width`
: Override width

`long height`
: Override height

`wstring name`
: The NAME attribute of the element (for backward compatibility).

OBJECT Element

Generic embedded object

Note: In principle, all properties on the object element are read-write but in some environments some properties may be read-only once the underlying object is instantiated.

Properties

`wstring classid`
: Identifies an implementation

`wstring align`
: Vertical or horizontal alignment

`wstring name`
: Submit as part of form

`wstring codeBase`
: Base URL for classid, data, archive

`wstring codeType`
: Content type for code

`wstring code`
: Code file

`wstring type`
: Content type for data

`long width`
: Initial width

`long height`
: Initial height

`wstring altHtml`
: HTML contained within the OBJECT tag that is displayed by user agents that cannot display the OBJECT itself.

`long vspace`
: Vertical gutter

`long hspace`
: Horizontal gutter

`long tabIndex`
: Position in tab order

`HTMLFormElement form`
: Returns the form the element is contained within. Returns null if not within the context of a form. (read-only).

`wstring data`
: The location of the object’s data.

Image Maps

MAP Element

Client side image map

Properties

`wstring name`
: Name, for reference by usemap

`NamedNodeList areas`
: Return a list of the areas in the map

AREA Element

Client side image map area

Properties

`wstring shape`
: Controls interpretation of coordinates

`wstring coords`
: Comma-separated list of lengths

`wstring href`
: URL for linked resource

`wstring target`
: Render in this frame

`wstring alt`
: Short description

`boolean noHref`
: This region has no action

`long tabIndex`
: Position in tab order

`wstring accessKey`
: Accessibility key character.

Table Elements

TABLE Element

The create* and delete* methods on the table provide easy access to construct and modify tables. HTML 4.0 defines that only 1 CAPTION, THEAD and TFOOT can exist in a table. If one exists, and the createTHead() or createTFoot() method is called, the method returns the THead or TFoot element that already exists.

Properties

`long cols`
: Number of columns

`wstring border`
: Border width

`wstring frame`
: Which parts of table frame to include

`wstring rules`
: Rulings between rows and cols

`long cellSpacing`
: Spacing between cells

`long cellPadding`
: Spacing within cells

`wstring bgColor`
: Background color for cells

`wstring align`
: Position of the table with respect to the document

`long width`
: The desired width of the table

Methods

`NamedNodeList getRows();`
: Get all row elements

`HTMLTableSection getTHead();`
: Select the table head

`HTMLTableSection getTFoot();`
: Select the table footer

`NamedNodeList getTBodies();`

`HTMLTableCaption getCaption();`
: Get the table caption

`HTMLElement createTHead();`

`void deleteTHead();`
: Delete the table header

`HTMLElement createTFoot();`
: Create a table footer

`void deleteTFoot();`
: Delete the table footer

`HTMLTableCaption createCaption();`
: Create a caption

`void deleteCaption();`
: Delete the caption

`HTMLElement insertRow(in long index);`
: Insert a new blank row

`void deleteRow(in long index);`
: Delete a row

CAPTION Element

Table Caption

Properties

`wstring align`
: Text alignment

COLGROUP Element

Properties

`long span`
: Used to delimit column groups

`wstring align`
: Horizontal alignment in cells

`wstring vAlign`
: Vertical alignment in cells

`long width`
: Default width

THEAD, TBODY, TFOOT Elements

Properties

`wstring align`
: Horizontal alignment in cells

`wstring vAlign`
: Vertical alignment in cells

Methods

```
NamedNodeList getRows()
HTMLElement insertRow(in long index);
void deleteRow(in long index);
```

TR Element

The rowIndex returns the index relative to the entire table, and sectionRowIndex returns the index relative to the section the element is scoped within (THead, TFoot, or TBody)

Properties

`wstring align`
: Horizontal alignment in cells

`wstring vAlign`
: Vertical alignment in cells

`wstring bgColor`
: Background color for rows

Methods

```
long getRowIndex();
long getSectionRowIndex();
NamedNodeList getCells();
HTMLElement insertCell(in long index);
void deleteCell(in long index);
```

TH, TD Element

Properties

`abbr`
: Abbreviation for header cells

`axis`
: Names group of related headers

`headers`
: List of ids for header cells

`scope`
: Scope covered by header cells

`long rowSpan`
: Number of rows spanned by cell

`long colSpan`
: Number of columns spanned by cell

`wstring align`
: Horizontal alignment in cells

`wstring vAlign`
: Vertical alignment in cells

`wstring bgColor`
: Cell background color

`boolean noWrap`
: Suppress word wrap

`long width`
: Width for cell

`long height`
: Height for cell

Methods

```
long getCellIndex();
```

Frame and related Elements

FRAMESET Element

Properties

`wstring rows`
: List of lengths (default 100%, 1 row)

`wstring cols`
: List of columns (default 100%, 1 col)

FRAME Element

Subwindow

Properties

`wstring src`
: Source URL

`wstring name`
: Name of frame, for targetting

`boolean noResize`
: Forbid user from resizing frame?

`wstring scrolling`
: Scrollbar or none

`wstring frameBorder`
: Request frame borders?

`long marginWidth`
: Margin width in pixels

`long marginHeight`
: Margin height in pixels

IFRAME Element

Inline subwindows

Properties

`wstring src`
: Source URL

`wstring name`
: Name, for targetting

`wstring scrolling`
: Scrollbar or none

`wstring align`
: Vertical or horizontal alignment

`long marginWidth;`
: Margin width in pixels

`long marginHeight;`
: Margin height in pixels

`long width`
: Frame width

`long height`
: Frame height

`long vspace`
: Vertical gutter

`long hspace`
: Horizontal gutter

Miscellaneous Elements

BR Element

Forced line break

Properties

`wstring clear`
: Control text flow

COMMENT Element

Properties

`wstring text`
: Comment text

SCRIPT Element

Script statements

Properties

`wstring src`
: URL of external script

`wstring text`
: The script contained by the element.

`wstring type`
: Content type of script language

`wstring language`
: Predefined script language name

`boolean defer`
: Indicates that the user agent can defer processing of the script.

HR Element

Horizontal rule

Properties

`wstring align`
: Text alignment

`boolean noShade`
: Indicates to the user agent that there should be no shading in the rendering of this element.

`long width`
: Width of rule

`long size`
: The height of the rule.

Appendix A: HTML IDL Interface definitions

This contains the complete IDL definitions for the HTML support provided by the Document Object Model.

```
//hb////-*-Mode: C++-*-///////////////////////////////////////////////////////////
//                                                                              //
// NAME        :                                                                //
// DESCRIPTION :                                                                //
// HISTORY     :                                                                //
//                                                                              //
//he//////////////////////////////////////////////////////////////////////////////



interface HTMLStyle {
  // Hook for later level
};

interface NamedNodeList {

  readonly attribute long length;
  Node item(in long index);
  Node namedItem(in wstring name);

};

interface HTMLDocument : Document
{

  // These are all methods, because they do some processing
  // of the document
  NamedNodeList  getImages();
  NamedNodeList  getApplets();
  NamedNodeList  getLinks();
  NamedNodeList  getForms();
  NamedNodeList  getAnchors();

  wstring         getCookie();
  wstring         setCookie(in wstring cookie);

  readonly attribute wstring       referrer;
  readonly attribute wstring       fileSize;
  readonly attribute wstring       fileCreatedDate;
  readonly attribute wstring       fileModifiedDate;
  readonly attribute wstring       fileUpdatedDate;
  readonly attribute wstring       title;
```

```
  readonly attribute wstring       domain;
  readonly attribute wstring       URL;

  attribute HTMLElement body;

  void      open();
  void      write(in wstring text);
  void      writeln(in wstring text);
  void      close();

};

interface HTMLElement : Element
{

  attribute wstring         className;
  attribute wstring         id;
  attribute HTMLStyle       style;
  attribute wstring         title;
  attribute wstring         lang;
  attribute wstring         dir;

};

interface HTMLLinkElement : HTMLElement
{

  attribute wstring href;
  attribute wstring rel;
  attribute wstring rev;
  attribute wstring type;
  attribute wstring media;
  attribute wstring charset;
  attribute wstring hreflang;
  attribute wstring target;
  attribute boolean disabled;

};

interface HTMLTitleElement : HTMLElement
{

  attribute wstring text;

};

interface HTMLMetaElement : HTMLElement
{

  attribute wstring httpEquiv;
  attribute wstring content;
  attribute wstring name;
  attribute wstring scheme;
};

interface HTMLBaseElement : HTMLElement
{
```

```
  attribute wstring href;
  attribute wstring target;
};

interface HTMLIsIndexElement : HTMLElement
{

  attribute wstring  prompt;

};

interface HTMLStyleElement : HTMLElement
{

  attribute wstring type;
  attribute boolean disabled;  attribute wstring media;

};

interface HTMLBodyElement : HTMLElement
{

  attribute wstring background;
  attribute wstring bgColor;
  attribute wstring text;
  attribute wstring link;
  attribute wstring vlink;
  attribute wstring alink;

};

interface HTMLFormElement : HTMLElement
{

  attribute wstring action;
  attribute wstring method;
  attribute wstring target;
  attribute wstring name;

  readonly attribute long    length;

  attribute wstring enctype;
  attribute wstring acceptCharset;  attribute NamedNodeList elements;

};

interface HTMLSelectElement : HTMLElement
{

  attribute long     selectedIndex;
  attribute long     size;
  attribute boolean  multiple;
  attribute wstring  name;

  readonly attribute wstring  type;
```

```
  attribute wstring  value;
  attribute boolean  disabled;
  attribute long     length;
  attribute long     tabIndex;
  readonly attribute HTMLFormElement  form;
  attribute NamedNodeList            options;

  void add(in HTMLElement element, in HTMLElement before);
  void remove(in long index);
  void blur();
  void focus();
};

interface HTMLOptionElement : HTMLElement
{

  attribute wstring         text;
  attribute long            index;
  attribute wstring         value;
  attribute boolean         defaultSelected;
  attribute boolean         disabled;

  attribute wstring         label;

  readonly attribute boolean         selected;
  readonly attribute HTMLFormElement form;

};

interface HTMLFieldSetElement : HTMLElement
{

  attribute wstring align;

};

interface HTMLLegendElement : HTMLElement
{

  attribute wstring align;

};

interface HTMLInputElement : HTMLElement
{

  readonly  attribute wstring type;
  attribute wstring  value;
  attribute wstring  name;
  attribute boolean  disabled;
  attribute wstring  defaultValue;
  attribute long     size;
  attribute long     maxLength;
  attribute boolean  readOnly;
  attribute boolean  checked;
  attribute boolean  defaultChecked;
  attribute wstring  src;
```

```
  attribute wstring  alt;
  attribute wstring  usemap;
  attribute wstring  align;
  attribute long     tabIndex;
  attribute wstring  accessKey;
  attribute wstring  accept;
  readonly attribute HTMLFormElement form;

  void blur();
  void focus();
  void select();
  void click();
};

interface HTMLTextAreaElement : HTMLElement
{

  readonly attribute wstring  type;
  attribute wstring  value;
  attribute wstring  name;
  attribute boolean  disabled;
  attribute wstring  defaultValue;
  attribute boolean  readOnly;
  attribute long     rows;
  attribute long     cols;
  attribute long     tabIndex;
  readonly attribute HTMLFormElement form;

  void blur();
  void focus();
  void select();
};

interface HTMLButtonElement : HTMLElement
{

  readonly attribute wstring  type;
  attribute wstring  value;
  attribute wstring  name;
  attribute boolean  disabled;
  attribute long     tabIndex;
  attribute wstring  accessKey;
  readonly attribute HTMLFormElement form;

};

interface HTMLLabelElement : HTMLElement
{

  attribute wstring htmlFor;
  attribute wstring accessKey;

};

interface HTMLAnchorElement : HTMLElement
{
```

```
  attribute wstring target;
  attribute wstring href;
  attribute wstring rel;
  attribute wstring rev;
  attribute wstring name;
  attribute wstring accessKey;
  attribute long    tabIndex;
  attribute wstring charset;

};

interface HTMLBaseFontElement : HTMLElement
{

  attribute wstring color;
  attribute wstring face;
  attribute long    size;

};

interface HTMLBRElement : HTMLElement
{

  attribute wstring clear;

};

interface HTMLImageElement : HTMLElement
{

  attribute boolean  isMap;
  attribute wstring  useMap;
  attribute wstring  border;
  attribute long     vspace;
  attribute long     hspace;
  attribute wstring  alt;
  attribute wstring  src;
  attribute wstring  lowSrc;
  attribute wstring  align;
  attribute long     width;
  attribute long     height;
  attribute wstring  name;

};

interface HTMLFontElement : HTMLElement
{

  attribute wstring color;
  attribute wstring face;
  attribute long    size;

};

interface HTMLModElement : HTMLElement
{
```

```
  attribute wstring cite;
  attribute wstring dateTime;

};

interface HTMLObjectElement : HTMLElement
{

  attribute wstring   classId;
  attribute wstring   data;
  attribute wstring   align;
  attribute wstring   name;
  attribute wstring   codeBase;
  attribute wstring   codeType;
  attribute wstring   code;
  attribute wstring   type;
  attribute long      width;
  attribute long      height;
  attribute wstring   altHtml;
  attribute long      vspace;
  attribute long      hspace;
  attribute long      tabIndex;

  readonly attribute HTMLFormElement form;

};

interface HTMLQuoteElement : HTMLElement
{

  attribute wstring cite;

};

interface HTMLScriptElement : HTMLElement
{

  attribute wstring  src;
  attribute wstring  text;
  attribute wstring  type;
  attribute wstring  language;
  attribute boolean  defer;
};

interface HTMLUListElement : HTMLElement
{

  attribute boolean compact;
  attribute wstring type;

};

interface HTMLOListElement : HTMLElement
{

  attribute boolean compact;
  attribute wstring type;
```

```
  attribute long    start;

};

interface HTMLLIElement : HTMLElement
{

  attribute wstring type;
  attribute long    value;

};

interface HTMLDListElement : HTMLElement {
  attribute boolean compact;

};

interface HTMLDivElement : HTMLElement
{

  attribute wstring align;

};

interface HTMLHRElement : HTMLElement
{

  attribute wstring  align;
  attribute boolean  noShade;
  attribute long     width;
  attribute long     size;

};

interface HTMLParagraphElement : HTMLElement
{

  attribute wstring align;

};

interface HTMLHeadingElement : HTMLElement
{

  attribute wstring align;

};

interface HTMLMapElement : HTMLElement
{

  readonly attribute NamedNodeList areas;
  attribute wstring name;

};

interface HTMLAreaElement : HTMLElement
```

```
{

  attribute wstring  shape;
  attribute wstring  coords;
  attribute wstring  href;
  attribute wstring  target;
  attribute wstring  alt;
  attribute boolean  noHref;
  attribute long     tabIndex;
  attribute wstring  accessKey;
};

interface HTMLTableCaption : HTMLElement
{

  attribute wstring align;

};

interface HTMLTable : HTMLElement
{

  attribute long                cols;
  attribute wstring             border;
  attribute wstring             frame;
  attribute wstring             rules;
  attribute long                cellSpacing;
  attribute long                cellPadding;
  attribute wstring             bgColor;
  attribute wstring             align;
  attribute long                width;
  attribute HTMLTableCaption   caption;
  readonly attribute HTMLTableSection   thead;
  readonly attribute HTMLTableSection   tfoot;

  NamedNodeList     getRows();
  NamedNodeList     getTBodies();

  HTMLElement        createTHead();
  void               deleteTHead();

  HTMLElement        createTFoot();
  void               deleteTFoot();

  HTMLTableCaption   createCaption();
  void               deleteCaption();

  HTMLElement        insertRow(in long index);
  void               deleteRow(in long index);

};

interface HTMLTableCol : HTMLElement
{

  attribute long     span;
  attribute wstring  align;
```

```
  attribute wstring  valign;
  attribute long     width;

};

interface HTMLTableSection : HTMLElement
{

  NamedNodeList getRows();
  HTMLElement insertRow(in long index);
  void        deleteRow(in long index);

  attribute wstring align;
  attribute wstring valign;

};

interface HTMLTableRow : HTMLElement
{

  attribute long     rowIndex;
  attribute long     sectionRowIndex;
  attribute wstring  align;
  attribute wstring  valign;
  attribute wstring  bgColor;

  NamedNodeList getCells();

  HTMLElement insertCell(in long index);
  void        deleteCell(in long index);

};

interface HTMLTableCell : HTMLElement
{

  attribute long     cellIndex;
  attribute long     rowSpan;
  attribute long     colSpan;
  attribute wstring  align;
  attribute wstring  valign;
  attribute wstring  bgColor;
  attribute boolean  noWrap;
  attribute long     width;
  attribute long     height;

};

interface HTMLCommentElement : HTMLElement
{

  attribute wstring text;

};

interface HTMLFrameSetElement : HTMLElement
{
```

```
  attribute wstring rows;
  attribute wstring cols;

};

interface HTMLFrameElement : HTMLElement
{

  attribute wstring  src;
  attribute wstring  name;
  attribute boolean  noResize;
  attribute wstring  scrolling;
  attribute wstring  frameBorder;
  attribute long     marginWidth;
  attribute long     marginHeight;

};

interface HTMLIFrameElement : HTMLElement
{

  attribute wstring  src;
  attribute wstring  name;
  attribute wstring  scrolling;
  attribute wstring  align;
  attribute long     marginWidth;
  attribute long     marginHeight;
  attribute long     width;
  attribute long     height;

};
```

Appendix B: Java HTML API definitions

The Java API definitions will be added in the next public release of the HTML DOM Level One.

Appendix C: ECMAScript HTML API definitions

This will contain the ECMAScript bindings for the HTML DOM.

Appendix A: Document Object Model Contributors

Note: *This specification was prepared and approved for publication by the W3C DOM Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval.*

Members of the DOM Activity contributing to this specification were:

Lauren Wood, SoftQuad, *chair*
Arnaud Le Hors, W3C, *W3C staff contact*
Andrew Watson, Object Management Group
Bill Smith, Sun
Chris Lovett, Microsoft
Chris Wilson, Microsoft
David Singer, IBM
Eric Vasilik, Microsoft
Gavin Nicol, INSO
James Clark, invited
Jared Sorensen, Novell
Jonathan Robie, Texcel
Mike Champion, ArborText
Paul Grosso, ArborText
Peter Sharpe, SoftQuad
Phil Karlton, Netscape
Rick Gessner, Netscape
Robert Sutor, IBM
Scott Isaacs, Microsoft
Sharon Adler, INSO
Steve Byrne, JavaSoft
Tim Bray, invited
Vidur Apparao, Netscape

Appendix B: Document Object Model Glossary

Editors

Robert Sutor, *IBM*

Several of the following term definitions have been borrowed or modified from similar definitions in other W3C documents. Links to these documents follow the definitions.

ancestor
: An ancestor node of any node A is any node above A in a tree model of a document, where "above" means "toward the root."

API
An API is an application programming interface, a set of functions or methods used to access some functionality.

child
A child is an immediate descendant node of a node.

client application
A [client] application is any software that uses the Document Object Model programming interfaces provided by the hosting implementation to accomplish useful work. Some examples of client applications are scripts within an HTML or XML document.

content model
The content model is a simple grammar governing the allowed types of the child elements and the order in which they appear. See [XML]

context
A context specifies an access pattern (or path): a set of interfaces which give you a way to interact with a model. For example, imagine a model with different colored arcs connecting data nodes. A context might be a sheet of colored acetate that is placed over the model allowing you a partial view of the total information in the model.

cooked model
A model for a document that represents the document after it has been manipulated in some way. For example, any combination of any of the following transformations would create a cooked model:

1. Expansion of internal text entities.
2. Expansion of external entities.
3. Model augmentation with style-specified generated text.
4. Execution of style-specified reordering.
5. Execution of scripts.

A browser might only be able to provide access to a cooked model, while an editor might provide access to a cooked or the initial structure model (also known as the uncooked model) for a document.

cursor
A cursor is an object representation of a node. It may possess information about context and the path traversed to reach the node.

deprecation
When new releases of specifications are released, some older features may be marked as being deprecated. This means that new work should not use the features and that although they are supported in the current release, they may not be supported or available in future releases.

descendant
A descendant node of any node A is any node below A in a tree model of a document, where "above" means "toward the root."

ECMAScript
The programming language defined by the ECMA-262 standard. As stated in the standard, the originating technology for ECMAScript was JavaScript. Note that in the ECMAScript binding, the word "property" is used in the same sense as the IDL term "attribute."

element
Each document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements by an empty-element tag. Each element has a type, identified by name, and may have a set of attributes. Each attribute has a name and a value. [XML]

event propagation, also known as event bubbling
: This is the idea that an event can affect one object and a set of related objects. Any of the potentially affected objects can block the event or substitute a different one (upward event propagation). The event is broadcast from the node at which it originates to every parent node.

hosting implementation
: A [hosting] implementation is a software module that provides an implementation of the DOM interfaces so that a client application can use them. Some examples of hosting implementations are browsers, editors and document repositories.

HTML
: The HyperText Markup Language (HTML) is a simple markup language used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of applications. [HTML 3.2] [HTML4.0]

IDL
: An Interface Definition Language (IDL) is used to define the interfaces for accessing and operating upon objects. Examples of IDLs are the Object Management Group's IDL , Microsoft's IDL , and Sun's Java IDL .

implementor
: Companies, organizations, and individuals that claim to support the Document Object Model as an API for their products.

initial structure model
: Also known as the raw structure model or the uncooked model, this represents the document before it has been modified by entity expansions, generated text, style-specified reordering, or the execution of scripts. In some implementations, this might correspond to the "initial parse tree" for the document, if it ever exists. Note that a given implementation might not be able to provide access to the initial structure model for a document, though an editor probably would.

language binding
: A programming language binding for an IDL specification is an implementation of the interfaces in the specification for the given language. For example, a Java language binding for the Document Object Model IDL specification would implement the concrete Java classes that provide the functionality exposed by the interfaces.

model
: A model is the actual data representation for the information at hand. Examples are the structural model and the style model representing the parse structure and the style information associated with a document. The model might be a tree, or a directed graph, or something else.

parent
: A parent is an immediate ancestor node of a node.

root node
: The root node is the unique node that is not a child of any other node. All other nodes are children or other descendents of the root node. [XML]

sibling
: Two nodes are siblings if they have the same parent node.

string comparison
: When string matching is required, it is to occur as though the comparison was between 2 sequences of code points from the Unicode 2.0 standard.

tag valid document
: A document is tag valid if all begin and end tags are properly balanced and nested.

type valid document
: A document is type valid if it conforms to an explicit DTD.

uncooked model
: See initial structure model.

well-formed document
: A document is well-formed if it is tag valid and entities are limited to single elements (i.e., single sub-trees).

XML
: Extensible Markup Language (XML) is an extremely simple dialect of SGML which is completely described in this document. The goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML. [XML]

Appendix C: Document Object Model Language Bindings

Appendix C.1: OMG IDL Definitions

http://www.w3.org/TR/WD-DOM/idl.tgz
http://www.w3.org/TR/WD-DOM/idl.zip

Appendix C.2: Java Bindings

http://www.w3.org/TR/WD-DOM/java.tgz
http://www.w3.org/TR/WD-DOM/java.zip
