Document Object Model (DOM) Level 2 Core Specification

Version 1.0

W3C Proposed Recommendation *27 September, 2000*

This version:
http://www.w3.org/TR/2000/PR-DOM-Level-2-Core-20000927
(PostScript file, PDF file, plain text, ZIP file)
Latest version:
http://www.w3.org/TR/DOM-Level-2-Core
Previous version:
http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510

Editors:
Mark Davis, *IBM*
Arnaud Le Hors, *W3C team contact until October 1999, then IBM*
Philippe Le Hégaret, *W3C, team contact (from November 1999)*
Jonathan Robie, *Texcel Research and Software AG*
Lauren Wood, *SoftQuad Software Inc., chair*

Abstract

This specification defines the Document Object Model Level 2 Core, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content and structure of documents. The Document Object Model Level 2 Core builds on the Document Object Model Level 1 Core.

The DOM Level 2 Core is made of a set of core interfaces to create and manipulate the structure and contents of a document. The Core also contains specialized interfaces dedicated to XML.

Status of this document

This is a W3C Proposed Recommendation for review by W3C members and other interested parties. W3C Advisory Committee Members are invited to send formal comments, visible only to the W3C Team, to dom-review@w3.org until October 25, 2000.

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

Publication as a Proposed Recommendation does not imply endorsement by the W3C membership. This is still a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite W3C Proposed Recommendations as other than "work in progress."

This document has been produced as part of the W3C DOM Activity. The authors of this document are the DOM WG members. Different modules of the Document Object Model have different editors.

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

Table of contents

Expanded Table of Contents

Copyright Notice

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

W3C Document Copyright Notice and License

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

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

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

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

1. A link or URL to the original W3C document.
2. The pre-existing copyright notice of the original author, or if it doesn’t exist, a notice of the form: "Copyright © [$date-of-document] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/" (Hypertext is preferred, but a textual representation is permitted.)
3. *If it exists*, the STATUS of the W3C document.

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

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

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

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

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

W3C Software Copyright Notice and License

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

Copyright © 1994-2000 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved.

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

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

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

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers. If none exist, then a notice of the following form: "Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/."
3. Notice of any changes or modifications to the W3C files, including the date changes were made. (We

What is the Document Object Model?

Editors
Jonathan Robie, Software AG

Introduction

The Document Object Model (DOM) is an application programming interface (*API* [p.101]) for valid *HTML* [p.102] and well-formed *XML* [p.103] documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM *interfaces* [p.102] for the XML internal and external subsets have not yet been specified.

As a W3C specification, one important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and *applications* [p.101] . The DOM is designed to be used with any programming language. In order to provide a precise, language-independent specification of the DOM interfaces, we have chosen to define the specifications in Object Management Group (OMG) IDL [OMGIDL], as defined in the CORBA 2.3.1 specification [CORBA]. In addition to the OMG IDL specification, we provide *language bindings* [p.102] for Java [Java] and ECMAScript [ECMAScript] (an industry-standard scripting language based on JavaScript [JavaScript] and JScript [JScript]).

Note: OMG IDL is used only as a language-independent and implementation-neutral way to specify *interfaces* [p.102] . Various other IDLs could have been used ([COM], [JavaIDL], [MIDL], ...). In general, IDLs are designed for specific computing environments. The Document Object Model can be implemented in any computing environment, and does not require the object binding runtimes generally associated with such IDLs.

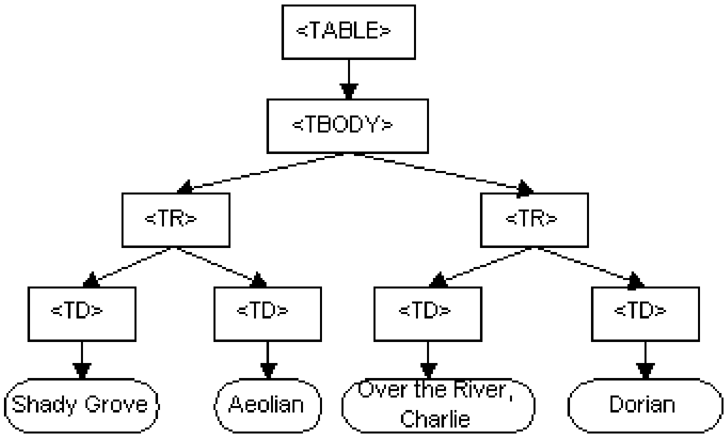
What the Document Object Model is

The DOM is a programming *API* [p.101] for documents. It is based on an object structure that closely resembles the structure of the documents it *models* [p.102] . For instance, consider this table, taken from an HTML document:

```
<TABLE>
<TBODY>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
</TR>
```

```
<TR>
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</TBODY>
</TABLE>
```

The DOM represents this table like this:



DOM representation of the example table

In the DOM, documents have a logical structure which is very much like a tree; to be more precise, which is like a "forest" or "grove", which can contain more than one tree. Each document contains zero or one doctype nodes, one root element node, and zero or more comments or processing instructions; the root element serves as the root of the element tree for the document. However, the DOM does not specify that documents must be *implemented* as a tree or a grove, nor does it specify how the relationships among objects be implemented. The DOM is a logical model that may be implemented in any convenient manner. In this specification, we use the term *structure model* to describe the tree-like representation of a document. We also use the term "tree" when referring to the arrangement of those information items which can be reached by using "tree-walking" methods; (this does not include attributes). One important property of DOM structure models is *structural isomorphism*: if any two Document Object Model implementations are used to create a representation of the same document, they will create the same structure model, in accordance with the XML Information Set [Infoset].

Note: There may be some variations depending on the parser being used to build the DOM. For instance, the DOM may not contain whitespaces in element content if the parser discards them.

The name "Document Object Model" was chosen because it is an "*object model* [p.103] " in the traditional object oriented design sense: documents are modeled using objects, and the model encompasses not only the structure of a document, but also the behavior of a document and the objects of which it is composed. In other words, the nodes in the above diagram do not represent a data structure, they represent objects, which have functions and identity. As an object model, the DOM identifies:

- the interfaces and objects used to represent and manipulate a document
- the semantics of these interfaces and objects - including both behavior and attributes
- the relationships and collaborations among these interfaces and objects

The structure of SGML documents has traditionally been represented by an abstract *data model* [p.101] , not by an object model. In an abstract *data model* [p.101] , the model is centered around the data. In object oriented programming languages, the data itself is encapsulated in objects that hide the data, protecting it from direct external manipulation. The functions associated with these objects determine how the objects may be manipulated, and they are part of the object model.

What the Document Object Model is not

This section is designed to give a more precise understanding of the DOM by distinguishing it from other systems that may seem to be like it.

- The Document Object Model is not a binary specification. DOM programs written in the same language binding will be source code compatible across platforms, but the DOM does not define any form of binary interoperability.
- The Document Object Model is not a way of persisting objects to XML or HTML. Instead of specifying how objects may be represented in XML, the DOM specifies how XML and HTML documents are represented as objects, so that they may be used in object oriented programs.
- The Document Object Model is not a set of data structures; it is an *object model* [p.103] that specifies interfaces. Although this document contains diagrams showing parent/child relationships, these are logical relationships defined by the programming interfaces, not representations of any particular internal data structures.
- The Document Object Model does not define what information in a document is relevant or how information in a document is structured. For XML, this is specified by the W3C XML Information Set [Infoset]. The DOM is simply an *API* [p.101] to this information set.
- The Document Object Model, despite its name, is not a competitor to the Component Object Model (COM). COM, like CORBA, is a language independent way to specify interfaces and objects; the DOM is a set of interfaces and objects designed for managing HTML and XML documents. The DOM may be implemented using language-independent systems like COM or CORBA; it may also be implemented using language-specific bindings like the Java or ECMAScript bindings specified in this document.

Where the Document Object Model came from

The DOM originated as a specification to allow JavaScript scripts and Java programs to be portable among Web browsers. "Dynamic HTML" was the immediate ancestor of the Document Object Model, and it was originally thought of largely in terms of browsers. However, when the DOM Working Group was formed at W3C, it was also joined by vendors in other domains, including HTML or XML editors and document repositories. Several of these vendors had worked with SGML before XML was developed; as a result, the DOM has been influenced by SGML Groves and the HyTime standard. Some of these vendors had also developed their own object models for documents in order to provide an API for SGML/XML editors or document repositories, and these object models have also influenced the DOM.

Entities and the DOM Core

In the fundamental DOM interfaces, 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 entity'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. Since numeric character references and pre-defined entities are not recognized as such in CDATA sections, or in the SCRIPT and STYLE elements in HTML, they are not replaced by the single character they appear to refer to. If the example above were enclosed in a CDATA section, the "&" would not be replaced by "&"; neither would the <p> be recognized as a start tag. The representation of general entities, both internal and external, are defined within the extended (XML) interfaces of DOM Level 1 [DOM Level 1].

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. Also, *implementations* [p.102] should be aware of the fact that serialization into a character encoding ("charset") that does not fully cover ISO 10646 may fail if there are characters in markup or CDATA sections that are not present in the encoding.

Compliance

The Document Object Model level 2 consists of several modules: Core, HTML, Views, StyleSheets, CSS, Events, Traversal, and Range. The DOM Core represents the functionality used for XML documents, and also serves as the basis for DOM HTML.

A compliant implementation of the DOM must implement all of the fundamental interfaces in the Core chapter with the semantics as defined. Further, it must implement at least one of the HTML DOM and the extended (XML) interfaces with the semantics as defined. The other modules are optional.

A DOM application can use the `hasFeature` *method* [p.102] of the `DOMImplementation` [p.22] interface to determine whether the module is supported or not. The feature strings for all modules in DOM Level 2 are listed in the following table; (strings are case-insensitive):

Module	Feature String
XML	XML
HTML	HTML
Views	Views
StyleSheets	StyleSheets
CSS	CSS
CSS (extended interfaces)	CSS2
Events	Events
User Interface Events (`UIEvent` interface)	UIEvents
Mouse Events (`MouseEvents` interface)	MouseEvents
Mutation Events (`MutationEvent` interface)	MutationEvents
HTML Events	HTMLEvents
Traversal	Traversal
Range	Range

The following table contains all dependencies between modules:

Module	Implies
Views	XML or HTML
StyleSheets	StyleSheets and XML or HTML
CSS	StyleSheets, Views and XML or HTML
CSS2	CSS, StyleSheets, Views and XML or HTML
Events	XML or HTML
UIEvents	Views, Events and XML or HTML
MouseEvents	UIEvents, Views, Events and XML or HTML
MutationEvents	Events and XML or HTML
HTMLEvents	Events and HTML
Traversal	XML or HTML
Range	XML or HTML

DOM Interfaces and DOM Implementations

The DOM specifies interfaces which may be used to manage XML or HTML documents. It is important to realize that these interfaces are an abstraction - much like "abstract base classes" in C++, they are a means of specifying a way to access and manipulate an application's internal representation of a document. Interfaces do not imply a particular concrete implementation. Each DOM application is free to maintain documents in any convenient representation, as long as the interfaces shown in this specification are supported. Some DOM implementations will be existing programs that use the DOM interfaces to access software written long before the DOM specification existed. Therefore, the DOM is designed to avoid implementation dependencies; in particular,

1. Attributes defined in the IDL do not imply concrete objects which must have specific data members - in the language bindings, they are translated to a pair of get()/set() functions, not to a data member. Read-only attributes have only a get() function in the language bindings.
2. DOM applications may provide additional interfaces and objects not found in this specification and still be considered DOM compliant.
3. Because we specify interfaces and not the actual objects that are to be created, the DOM cannot know what constructors to call for an implementation. In general, DOM users call the createX() methods on the Document class to create document structures, and DOM implementations create their own internal representations of these structures in their implementations of the createX() functions.

The Level 1 interfaces were extended to provide both Level 1 and Level 2 functionality.

DOM implementations in languages other than Java or ECMA Script may choose bindings that are appropriate and natural for their language and run time environment. For example, some systems may need to create a Document2 class which inherits from Document and contains the new methods and attributes.

DOM Level 2 does not specify multithreading mechanisms.

1. Document Object Model Core

Editors

Arnaud Le Hors, IBM
Mike Champion, ArborText (for DOM Level 1 from November 20, 1997)
Steve Byrne, JavaSoft (for DOM Level 1 until November 19, 1997)
Gavin Nicol, Inso EPS (for DOM Level 1)
Lauren Wood, SoftQuad, Inc. (for DOM Level 1)

1.1. Overview of the DOM Core Interfaces

This section defines a set of objects and interfaces for accessing and manipulating document objects. The functionality specified in this section (the *Core* functionality) is sufficient to allow software developers and web script authors to access and manipulate parsed HTML and XML content inside conforming products. The DOM Core API also allows creation and population of a `Document` [p.25] object using only DOM API calls; loading a `Document` and saving it persistently is left to the product that implements the DOM API.

1.1.1. The DOM Structure Model

The DOM presents documents as a hierarchy of `Node` [p.35] objects that also implement other, more specialized interfaces. Some types of nodes may have *child* [p.101] nodes of various types, and others are leaf nodes that cannot have anything below them in the document structure. For XML and HTML, the node types, and which node types they may have as children, are as follows:

- `Document` [p.25] -- `Element` [p.55] (maximum of one), `ProcessingInstruction` [p.68] , `Comment` [p.64] , `DocumentType` [p.65] (maximum of one)
- `DocumentFragment` [p.24] -- `Element` [p.55] , `ProcessingInstruction` [p.68] , `Comment` [p.64] , `Text` [p.63] , `CDATASection` [p.64] , `EntityReference` [p.68]
- `DocumentType` [p.65] -- no children
- `EntityReference` [p.68] -- `Element` [p.55] , `ProcessingInstruction` [p.68] , `Comment` [p.64] , `Text` [p.63] , `CDATASection` [p.64] , `EntityReference`
- `Element` [p.55] -- `Element`, `Text` [p.63] , `Comment` [p.64] , `ProcessingInstruction` [p.68] , `CDATASection` [p.64] , `EntityReference` [p.68]
- `Attr` [p.53] -- `Text` [p.63] , `EntityReference` [p.68]
- `ProcessingInstruction` [p.68] -- no children
- `Comment` [p.64] -- no children
- `Text` [p.63] -- no children
- `CDATASection` [p.64] -- no children
- `Entity` [p.67] -- `Element` [p.55] , `ProcessingInstruction` [p.68] , `Comment` [p.64] , `Text` [p.63] , `CDATASection` [p.64] , `EntityReference` [p.68]
- `Notation` [p.66] -- no children

The DOM also specifies a `NodeList` [p.44] interface to handle ordered lists of `Nodes` [p.35] , such as the children of a `Node` [p.35] , or the *elements* [p.102] returned by the `getElementsByTagName` method of the `Element` [p.55] interface, and also a `NamedNodeMap` [p.45] interface to handle unordered sets of nodes referenced by their name attribute, such as the attributes of an `Element`. `NodeList` [p.44] and `NamedNodeMap` [p.45] objects in the DOM are *live*; that is, changes to the underlying document structure are reflected in all relevant `NodeList` and `NamedNodeMap` objects. For example, if a DOM user gets a `NodeList` object containing the children of an `Element` [p.55] , then subsequently adds more children to that *element* [p.102] (or removes children, or modifies them), those changes are automatically reflected in the `NodeList`, without further action on the user's part. Likewise, changes to a `Node` [p.35] in the tree are reflected in all references to that `Node` in `NodeList` and `NamedNodeMap` objects.

Finally, the interfaces `Text` [p.63] , `Comment` [p.64] , and `CDATASection` [p.64] all inherit from the `CharacterData` [p.49] interface.

1.1.2. Memory Management

Most of the APIs defined by this specification are *interfaces* rather than classes. That means that an implementation need only expose methods with the defined names and specified operation, not implement classes that correspond directly to the interfaces. This allows the DOM APIs to be implemented as a thin veneer on top of legacy applications with their own data structures, or on top of newer applications with different class hierarchies. This also means that ordinary constructors (in the Java or C++ sense) cannot be used to create DOM objects, since the underlying objects to be constructed may have little relationship to the DOM interfaces. The conventional solution to this in object-oriented design is to define *factory* methods that create instances of objects that implement the various interfaces. Objects implementing some interface "X" are created by a "createX()" method on the `Document` [p.25] interface; this is because all DOM objects live in the context of a specific Document.

The DOM Level 2 API does *not* define a standard way to create `DOMImplementation` [p.22] objects; DOM implementations must provide some proprietary way of bootstrapping these DOM interfaces, and then all other objects can be built from there.

The Core DOM APIs are designed to be compatible with a wide range of languages, including both general-user scripting languages and the more challenging languages used mostly by professional programmers. Thus, the DOM APIs need to operate across a variety of memory management philosophies, from language bindings that do not expose memory management to the user at all, through those (notably Java) that provide explicit constructors but provide an automatic garbage collection mechanism to automatically reclaim unused memory, to those (especially C/C++) that generally require the programmer to explicitly allocate object memory, track where it is used, and explicitly free it for re-use. To ensure a consistent API across these platforms, the DOM does not address memory management issues at all, but instead leaves these for the implementation. Neither of the explicit language bindings devised by the DOM Working Group (for *ECMAScript* [p.102] and Java) require any memory management methods, but DOM bindings for other languages (especially C or C++) may require such support. These extensions will be the responsibility of those adapting the DOM API to a specific language, not the DOM Working Group.

1.1.3. Naming Conventions

While it would be nice to have attribute and method names that are short, informative, internally consistent, and familiar to users of similar APIs, the names also should not clash with the names in legacy APIs supported by DOM implementations. Furthermore, both OMG IDL and `ECMAScript` have significant limitations in their ability to disambiguate names from different namespaces that make it difficult to avoid naming conflicts with short, familiar names. So, some DOM names tend to be long and quite descriptive in order to be unique across all environments.

The Working Group has also attempted to be internally consistent in its use of various terms, even though these may not be common distinctions in other APIs. For example, we use the method name "remove" when the method changes the structural model, and the method name "delete" when the method gets rid of something inside the structure model. The thing that is deleted is not returned. The thing that is removed may be returned, when it makes sense to return it.

1.1.4. Inheritance vs. Flattened Views of the API

The DOM Core *APIs* [p.101] present two somewhat different sets of interfaces to an XML/HTML document: one presenting an "object oriented" approach with a hierarchy of *inheritance* [p.102] , and a "simplified" view that allows all manipulation to be done via the `Node` [p.35] interface without requiring casts (in Java and other C-like languages) or query interface calls in *COM* [p.101] environments. These operations are fairly expensive in Java and COM, and the DOM may be used in performance-critical environments, so we allow significant functionality using just the `Node` interface. Because many other users will find the *inheritance* [p.102] hierarchy easier to understand than the "everything is a `Node`" approach to the DOM, we also support the full higher-level interfaces for those who prefer a more object-oriented *API* [p.101] .

In practice, this means that there is a certain amount of redundancy in the *API* [p.101] . The Working Group considers the "*inheritance* [p.102] " approach the primary view of the API, and the full set of functionality on `Node` [p.35] to be "extra" functionality that users may employ, but that does not eliminate the need for methods on other interfaces that an object-oriented analysis would dictate. (Of course, when the O-O analysis yields an attribute or method that is identical to one on the `Node` interface, we don't specify a completely redundant one.) Thus, even though there is a generic `nodeName` attribute on the `Node` interface, there is still a `tagName` attribute on the `Element` [p.55] interface; these two attributes must contain the same value, but the Working Group considers it worthwhile to support both, given the different constituencies the DOM *API* [p.101] must satisfy.

1.1.5. The `DOMString` type

To ensure interoperability, the DOM specifies the following:

- **Type Definition *DOMString***

  A `DOMString` [p.17] is a sequence of *16-bit units* [p.101] .

IDL Definition

```
typedef sequence<unsigned short> DOMString;
```

- Applications must encode `DOMString` [p.17] using UTF-16 (defined in [Unicode] and Amendment 1 of [ISO/IEC 10646]).
  The UTF-16 encoding was chosen because of its widespread industry practice. Note that for both HTML and XML, the document character set (and therefore the notation of numeric character references) is based on UCS [ISO-10646]. A single numeric character reference in a source document may therefore in some cases correspond to two 16-bit units in a `DOMString` [p.17] (a high surrogate and a low surrogate).

  Note: Even though the DOM defines the name of the string type to be `DOMString` [p.17] , bindings may use different names. For example for Java, `DOMString` is bound to the `String` type because it also uses UTF-16 as its encoding.

Note: As of August 2000, the OMG IDL specification ([OMGIDL]) included a `wstring` type. However, that definition did not meet the interoperability criteria of the DOM *API* [p.101] since it relied on negotiation to decide the width and encoding of a character.

1.1.6. The `DOMTimeStamp` type

To ensure interoperability, the DOM specifies the following:

- **Type Definition** ***DOMTimeStamp***

  A `DOMTimeStamp` [p.18] represents a number of milliseconds.
  IDL Definition

  ```
  typedef unsigned long long DOMTimeStamp;
  ```

- **Note:** Even though the DOM uses the type `DOMTimeStamp` [p.18] , bindings may use different types. For example for Java, `DOMTimeStamp` is bound to the `long` type. In ECMAScript, `TimeStamp` is bound to the `Date` type because the range of the `integer` type is too small.

1.1.7. String comparisons in the DOM

The DOM has many interfaces that imply string matching. HTML processors generally assume an uppercase (less often, lowercase) normalization of names for such things as *elements* [p.102] , while XML is explicitly case sensitive. For the purposes of the DOM, string matching is performed purely by binary *comparison* [p.103] of the *16-bit units* [p.101] of the `DOMString` [p.17] . In addition, the DOM assumes that any case normalizations take place in the processor, *before* the DOM structures are built.

Note: Besides case folding, there are additional normalizations that can be applied to text. The W3C I18N Working Group is in the process of defining exactly which normalizations are necessary, and where they should be applied. The W3C I18N Working Group expects to require early normalization, which means that data read into the DOM is assumed to already be normalized. The DOM and applications built on top of it in this case only have to assure that text remains normalized when being changed. For further details, please see [Charmod].

1.1.8. XML Namespaces

The DOM Level 2 supports XML namespaces [Namespaces] by augmenting several interfaces of the DOM Level 1 Core to allow creating and manipulating *elements* [p.102] and attributes associated to a namespace.

As far as the DOM is concerned, special attributes used for declaring *XML namespaces* [p.103] are still exposed and can be manipulated just like any other attribute. However, nodes are permanently bound to *namespace URIs* [p.103] as they get created. Consequently, moving a node within a document, using the DOM, in no case results in a change of its *namespace prefix* [p.103] or namespace URI. Similarly, creating a node with a namespace prefix and namespace URI, or changing the namespace prefix of a node, does not result in any addition, removal, or modification of any special attributes for declaring the appropriate XML namespaces. Namespace validation is not enforced; the DOM application is responsible. In particular, since the mapping between prefixes and namespace URIs is not enforced, in general, the resulting document cannot be serialized naively. For example, applications may have to declare every namespace in use when serializing a document.

DOM Level 2 doesn't perform any URI normalization or canonicalization. The URIs given to the DOM are assumed to be valid (e.g., characters such as whitespaces are properly escaped), and no lexical checking is performed. Absolute URI references are treated as strings and *compared literally* [p.103] . How relative namespace URI references are treated is undefined. To ensure interoperability only absolute namespace URI references (i.e., URI references beginning with a scheme name and a colon) should be used. Note that because the DOM does no lexical checking, the empty string will be treated as a real namespace URI in DOM Level 2 methods. Applications must use the value `null` as the namespaceURI parameter for methods if they wish to have no namespace.

Note: In the DOM, all namespace declaration attributes are *by definition* bound to the namespace URI: "http://www.w3.org/2000/xmlns/". These are the attributes whose *namespace prefix* [p.103] or *qualified name* [p.103] is "xmlns". Although, at the time of writing, this is not part of the XML Namespaces specification [Namespaces], it is planned to be incorporated in a future revision.

In a document with no namespaces, the *child* [p.101] list of an `EntityReference` [p.68] node is always the same as that of the corresponding `Entity` [p.67] . This is not true in a document where an entity contains unbound *namespace prefixes* [p.103] . In such a case, the *descendants* [p.101] of the corresponding `EntityReference` nodes may be bound to different *namespace URIs* [p.103] , depending on where the entity references are. Also, because, in the DOM, nodes always remain bound to the same namespace URI, moving such `EntityReference` nodes can lead to documents that cannot be serialized. This is also true when the DOM Level 1 method `createEntityReference` of the `Document` [p.25] interface is used to create entity references that correspond to such entities, since the *descendants* [p.101] of the returned `EntityReference` are unbound. The DOM Level 2 does not support any mechanism to resolve namespace prefixes. For all of these reasons, use of such entities and entity references should be avoided or used with extreme care. A future Level of the DOM may include some additional support for handling these.

The new methods, such as `createElementNS` and `createAttributeNS` of the `Document` [p.25] interface, are meant to be used by namespace aware applications. Simple applications that do not use namespaces can use the DOM Level 1 methods, such as `createElement` and `createAttribute`. Elements and attributes created in this way do not have any namespace prefix, namespace URI, or local name.

Note: DOM Level 1 methods are namespace ignorant. Therefore, while it is safe to use these methods when not dealing with namespaces, using them and the new ones at the same time should be avoided. DOM Level 1 methods solely identify attribute nodes by their `nodeName`. On the contrary, the DOM Level 2 methods related to namespaces, identify attribute nodes by their `namespaceURI` and `localName`. Because of this fundamental difference, mixing both sets of methods can lead to unpredictable results. In particular, using `setAttributeNS`, an *element* [p.102] may have two attributes (or more) that have the same `nodeName`, but different `namespaceURI`s. Calling `getAttribute` with that `nodeName` could then return any of those attributes. The result depends on the implementation. Similarly, using `setAttributeNode`, one can set two attributes (or more) that have different `nodeName`s but the same `prefix` and `namespaceURI`. In this case `getAttributeNodeNS` will return either attribute, in an implementation dependent manner. The only guarantee in such cases is that all methods that access a named item by its `nodeName` will access the same item, and all methods which access a node by its URI and local name will access the same node. For instance, `setAttribute` and `setAttributeNS` affect the node that `getAttribute` and `getAttributeNS`, respectively, return.

1.2. Fundamental Interfaces

The interfaces within this section are considered *fundamental*, and must be fully implemented by all conforming implementations of the DOM, including all HTML DOM implementations [DOM Level 2 HTML], unless otherwise specified.

Exception *DOMException*

DOM operations only raise exceptions in "exceptional" circumstances, i.e., when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable). In general, DOM methods return specific error values in ordinary processing situations, such as out-of-bound errors when using `NodeList` [p.44] .

Implementations may raise other exceptions under other circumstances. For example, implementations may raise an implementation-dependent exception if a `null` argument is passed.

Some languages and object systems do not support the concept of exceptions. For such systems, error conditions may be indicated using native error reporting mechanisms. For some bindings, for example, methods may return error codes similar to those listed in the corresponding method descriptions.

IDL Definition

```
exception DOMException {
  unsigned short   code;
};
// ExceptionCode
```

```
const unsigned short      INDEX_SIZE_ERR                 = 1;
const unsigned short      DOMSTRING_SIZE_ERR             = 2;
const unsigned short      HIERARCHY_REQUEST_ERR          = 3;
const unsigned short      WRONG_DOCUMENT_ERR             = 4;
const unsigned short      INVALID_CHARACTER_ERR          = 5;
const unsigned short      NO_DATA_ALLOWED_ERR            = 6;
const unsigned short      NO_MODIFICATION_ALLOWED_ERR    = 7;
const unsigned short      NOT_FOUND_ERR                  = 8;
const unsigned short      NOT_SUPPORTED_ERR              = 9;
const unsigned short      INUSE_ATTRIBUTE_ERR            = 10;
// Introduced in DOM Level 2:
const unsigned short      INVALID_STATE_ERR              = 11;
// Introduced in DOM Level 2:
const unsigned short      SYNTAX_ERR                     = 12;
// Introduced in DOM Level 2:
const unsigned short      INVALID_MODIFICATION_ERR       = 13;
// Introduced in DOM Level 2:
const unsigned short      NAMESPACE_ERR                  = 14;
// Introduced in DOM Level 2:
const unsigned short      INVALID_ACCESS_ERR             = 15;
```

Definition group *ExceptionCode*

An integer indicating the type of error generated.

Note: Other numeric codes are reserved for W3C for possible future use.

Defined Constants

`DOMSTRING_SIZE_ERR`
: If the specified range of text does not fit into a DOMString

`HIERARCHY_REQUEST_ERR`
: If any node is inserted somewhere it doesn't belong

`INDEX_SIZE_ERR`
: If index or size is negative, or greater than the allowed value

`INUSE_ATTRIBUTE_ERR`
: If an attempt is made to add an attribute that is already in use elsewhere

`INVALID_ACCESS_ERR`, introduced in **DOM Level 2**.
: If a parameter or an operation is not supported by the underlying object.

`INVALID_CHARACTER_ERR`
: If an invalid or illegal character is specified, such as in a name. See *production 2* in the XML specification for the definition of a legal character, and *production 5* for the definition of a legal name character.

`INVALID_MODIFICATION_ERR`, introduced in **DOM Level 2**.
: If an attempt is made to modify the type of the underlying object.

`INVALID_STATE_ERR`, introduced in **DOM Level 2**.
If an attempt is made to use an object that is not, or is no longer, usable.

`NAMESPACE_ERR`, introduced in **DOM Level 2**.
If an attempt is made to create or change an object in a way which is incorrect with regard to namespaces.

`NOT_FOUND_ERR`
If an attempt is made to reference a node in a context where it does not exist

`NOT_SUPPORTED_ERR`
If the implementation does not support the type of object requested

`NO_DATA_ALLOWED_ERR`
If data is specified for a node which does not support data

`NO_MODIFICATION_ALLOWED_ERR`
If an attempt is made to modify an object where modifications are not allowed

`SYNTAX_ERR`, introduced in **DOM Level 2**.
If an invalid or illegal string is specified.

`WRONG_DOCUMENT_ERR`
If a node is used in a different document than the one that created it (that doesn't support it)

Interface *DOMImplementation*

The `DOMImplementation` interface provides a number of methods for performing operations that are independent of any particular instance of the document object model.

IDL Definition

```
interface DOMImplementation {
  boolean            hasFeature(in DOMString feature,
                                in DOMString version);
  // Introduced in DOM Level 2:
  DocumentType       createDocumentType(in DOMString qualifiedName,
                                        in DOMString publicId,
                                        in DOMString systemId)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Document           createDocument(in DOMString namespaceURI,
                                    in DOMString qualifiedName,
                                    in DocumentType doctype)
                                        raises(DOMException);
};
```

Methods

`createDocument` introduced in **DOM Level 2**
Creates an XML `Document` [p.25] object of the specified type with its document element. HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the document element to create.

`qualifiedName` of type `DOMString`
The *qualified name* [p.103] of the document element to be created.

`doctype` of type `DocumentType` [p.65]
The type of document to be created or `null`.
When `doctype` is not `null`, its `Node.ownerDocument` [p.39] attribute is set to the document being created.

Return Value

`Document` [p.25]	A new `Document` object.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from "http://www.w3.org/XML/1998/namespace" [Namespaces]. WRONG_DOCUMENT_ERR: Raised if `doctype` has already been used with a different document or was created from a different implementation.

`createDocumentType` introduced in **DOM Level 2**
Creates an empty `DocumentType` [p.65] node. Entity declarations and notations are not made available. Entity reference expansions and default attribute additions do not occur. It is expected that a future version of the DOM will provide a way for populating a `DocumentType`.
HTML-only DOM implementations do not need to implement this method.
Parameters
`qualifiedName` of type `DOMString` [p.17]
The *qualified name* [p.103] of the document type to be created.

`publicId` of type `DOMString`
The external subset public identifier.

`systemId` of type `DOMString`
The external subset system identifier.

Return Value

`DocumentType` [p.65]	A new `DocumentType` node with `Node.ownerDocument` [p.39] set to `null`.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed.

`hasFeature`
Test if the DOM implementation implements a specific feature.
Parameters
`feature` of type `DOMString` [p.17]
The name of the feature to test (case-insensitive). The values used by DOM features are defined throughout the DOM Level 2 specifications and listed in the Compliance [p.12] section. The name must be an *XML name* [p.103] . To avoid possible conflicts, as a convention, names referring to features defined outside the DOM specification should be made unique by reversing the name of the Internet domain name of the person (or the organization that the person belongs to) who defines the feature, component by component, and using this as a prefix. For instance, the W3C SVG Working Group defines the feature "org.w3c.dom.svg".

`version` of type `DOMString`
This is the version number of the feature to test. In Level 2, the string can be either "2.0" or "1.0". If the version is not specified, supporting any version of the feature causes the method to return `true`.

Return Value

`boolean`	`true` if the feature is implemented in the specified version, `false` otherwise.

No Exceptions

Interface *DocumentFragment*

`DocumentFragment` is a "lightweight" or "minimal" `Document` [p.25] object. It is very common to want to be able to extract a portion of a document's tree or to create a new fragment of a document. Imagine implementing a user command like cut or rearranging a document by moving fragments around. It is desirable to have an object which can hold such fragments and it is quite natural to use a Node for this purpose. While it is true that a `Document` object could fulfill this role, a `Document` object can potentially be a heavyweight object, depending on the underlying implementation. What is really needed for this is a very lightweight object. `DocumentFragment` is such an object.

Furthermore, various operations -- such as inserting nodes as children of another `Node` [p.35] -- may take `DocumentFragment` objects as arguments; this results in all the child nodes of the `DocumentFragment` being moved to the child list of this node.

The children of a `DocumentFragment` node are zero or more nodes representing the tops of any sub-trees defining the structure of the document. `DocumentFragment` nodes do not need to be *well-formed XML documents* [p.103] (although they do need to follow the rules imposed upon well-formed XML parsed entities, which can have multiple top nodes). For example, a `DocumentFragment` might have only one child and that child node could be a `Text` [p.63] node. Such a structure model represents neither an HTML document nor a well-formed XML document.

When a `DocumentFragment` is inserted into a `Document` [p.25] (or indeed any other `Node` [p.35] that may take children) the children of the `DocumentFragment` and not the `DocumentFragment` itself are inserted into the `Node`. This makes the `DocumentFragment` very useful when the user wishes to create nodes that are *siblings* [p.103] ; the `DocumentFragment` acts as the parent of these nodes so that the user can use the standard methods from the `Node` interface, such as `insertBefore` and `appendChild`.

IDL Definition

```
interface DocumentFragment : Node {
};
```

Interface *Document*

The `Document` interface represents the entire HTML or XML document. Conceptually, it is the *root* [p.103] of the document tree, and provides the primary access to the document's data.

Since elements, text nodes, comments, processing instructions, etc. cannot exist outside the context of a `Document`, the `Document` interface also contains the factory methods needed to create these objects. The `Node` [p.35] objects created have a `ownerDocument` attribute which associates them with the `Document` within whose context they were created.

IDL Definition

```
interface Document : Node {
  readonly attribute DocumentType      doctype;
  readonly attribute DOMImplementation  implementation;
  readonly attribute Element           documentElement;
  Element              createElement(in DOMString tagName)
                                        raises(DOMException);
  DocumentFragment   createDocumentFragment();
  Text               createTextNode(in DOMString data);
```

```
  Comment             createComment(in DOMString data);
  CDATASection        createCDATASection(in DOMString data)
                                        raises(DOMException);
  ProcessingInstruction createProcessingInstruction(in DOMString target,
                                                    in DOMString data)
                                        raises(DOMException);
  Attr                createAttribute(in DOMString name)
                                        raises(DOMException);
  EntityReference     createEntityReference(in DOMString name)
                                        raises(DOMException);
  NodeList            getElementsByTagName(in DOMString tagname);
  // Introduced in DOM Level 2:
  Node                importNode(in Node importedNode,
                                 in boolean deep)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Element             createElementNS(in DOMString namespaceURI,
                                      in DOMString qualifiedName)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Attr                createAttributeNS(in DOMString namespaceURI,
                                        in DOMString qualifiedName)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  NodeList            getElementsByTagNameNS(in DOMString namespaceURI,
                                             in DOMString localName);
  // Introduced in DOM Level 2:
  Element             getElementById(in DOMString elementId);
};
```

Attributes

`doctype` of type `DocumentType` [p.65] , readonly

The Document Type Declaration (see `DocumentType` [p.65]) associated with this document. For HTML documents as well as XML documents without a document type declaration this returns `null`. The DOM Level 2 does not support editing the Document Type Declaration. `docType` cannot be altered in any way, including through the use of methods inherited from the `Node` [p.35] interface, such as `insertNode` or `removeNode`.

`documentElement` of type `Element` [p.55] , readonly

This is a *convenience* [p.101] attribute that allows direct access to the child node that is the root element of the document. For HTML documents, this is the element with the tagName "HTML".

`implementation` of type `DOMImplementation` [p.22] , readonly

The `DOMImplementation` [p.22] object that handles this document. A DOM application may use objects from multiple implementations.

Methods

`createAttribute`

Creates an `Attr` [p.53] of the given name. Note that the `Attr` instance can then be set on an `Element` [p.55] using the `setAttributeNode` method.
To create an attribute with a qualified name and namespace URI, use the

`createAttributeNS` method.
Parameters
`name` of type `DOMString` [p.17]
The name of the attribute.

Return Value

`Attr` [p.53]	A new `Attr` object with the `nodeName` attribute set to `name`, and `localName`, `prefix`, and `namespaceURI` set to `null`. The value of the attribute is the empty string.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

`createAttributeNS` introduced in **DOM Level 2**
Creates an attribute of the given qualified name and namespace URI. HTML-only DOM implementations do not need to implement this method.
Parameters
`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the attribute to create.

`qualifiedName` of type `DOMString`
The *qualified name* [p.103] of the attribute to instantiate.

Return Value

Attr [p.53]	A new Attr object with the following attributes:

Attribute	Value
Node.nodeName [p.39]	qualifiedName
Node.namespaceURI [p.39]	namespaceURI
Node.prefix [p.40]	prefix, extracted from qualifiedName, or null if there is no prefix
Node.localName [p.38]	*local name*, extracted from qualifiedName
Attr.name [p.54]	qualifiedName
Node.nodeValue [p.39]	the empty string

Exceptions

DOMException [p.20]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the qualifiedName is malformed, if the qualifiedName has a prefix and the namespaceURI is null, if the qualifiedName has a prefix that is "xml" and the namespaceURI is different from "http://www.w3.org/XML/1998/namespace", or if the qualifiedName is "xmlns" and the namespaceURI is different from "http://www.w3.org/2000/xmlns/".

createCDATASection

Creates a CDATASection [p.64] node whose value is the specified string.

Parameters

data of type DOMString [p.17]

The data for the CDATASection [p.64] contents.

Return Value

CDATASection [p.64]	The new CDATASection object.

Exceptions

DOMException [p.20]	NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

createComment
Creates a Comment [p.64] node given the specified string.
Parameters
data of type DOMString [p.17]
The data for the node.

Return Value

Comment [p.64]	The new Comment object.

No Exceptions

createDocumentFragment
Creates an empty DocumentFragment [p.24] object.
Return Value

DocumentFragment [p.24]	A new DocumentFragment.

No Parameters
No Exceptions

createElement
Creates an element of the type specified. Note that the instance returned implements the Element [p.55] interface, so attributes can be specified directly on the returned object.
In addition, if there are known attributes with default values, Attr [p.53] nodes representing them are automatically created and attached to the element.
To create an element with a qualified name and namespace URI, use the createElementNS method.
Parameters
tagName of type DOMString [p.17]
The name of the element type to instantiate. For XML, this is case-sensitive. For HTML, the tagName parameter may be provided in any case, but it must be mapped to the canonical uppercase form by the DOM implementation.

Return Value

Element [p.55]	A new Element object with the nodeName attribute set to tagName, and localName, prefix, and namespaceURI set to null.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

`createElementNS` introduced in **DOM Level 2**

Creates an element of the given qualified name and namespace URI. HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]

The *namespace URI* [p.103] of the element to create.

`qualifiedName` of type `DOMString`

The *qualified name* [p.103] of the element type to instantiate.

Return Value

`Element` [p.55] A new `Element` object with the following attributes:

Attribute	Value
`Node.nodeName` [p.39]	`qualifiedName`
`Node.namespaceURI` [p.39]	`namespaceURI`
`Node.prefix` [p.40]	prefix, extracted from `qualifiedName`, or `null` if there is no prefix
`Node.localName` [p.38]	*local name*, extracted from `qualifiedName`
`Element.tagName` [p.56]	`qualifiedName`

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from "http://www.w3.org/XML/1998/namespace" [Namespaces].

`createEntityReference`

Creates an `EntityReference` [p.68] object. In addition, if the referenced entity is known, the child list of the `EntityReference` node is made the same as that of the corresponding `Entity` [p.67] node.

Note: If any descendant of the `Entity` [p.67] node has an unbound *namespace prefix* [p.103] , the corresponding descendant of the created `EntityReference` [p.68] node is also unbound; (its `namespaceURI` is `null`). The DOM Level 2 does not support any mechanism to resolve namespace prefixes.

Parameters

`name` of type `DOMString` [p.17]

The name of the entity to reference.

Return Value

`EntityReference` [p.68]	The new `EntityReference` object.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createProcessingInstruction`

Creates a `ProcessingInstruction` [p.68] node given the specified name and data strings.

Parameters

`target` of type `DOMString` [p.17]

The target part of the processing instruction.

`data` of type `DOMString`

The data for the node.

Return Value

`ProcessingInstruction` [p.68]	The new `ProcessingInstruction` object.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified target contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createTextNode`

Creates a `Text` [p.63] node given the specified string.

Parameters

`data` of type `DOMString` [p.17]

The data for the node.

Return Value

`Text` [p.63]	The new `Text` object.

No Exceptions

`getElementById` introduced in **DOM Level 2**

Returns the `Element` [p.55] whose `ID` is given by `elementId`. If no such element exists, returns `null`. Behavior is not defined if more than one element has this `ID`.

Note: The DOM implementation must have information that says which attributes are of type ID. Attributes with the name "ID" are not of type ID unless so defined. Implementations that do not know whether attributes are of type ID or not are expected to return `null`.

Parameters

`elementId` of type `DOMString` [p.17]

The unique `id` value for an element.

Return Value

`Element` [p.55]	The matching element.

No Exceptions

`getElementsByTagName`

Returns a `NodeList` [p.44] of all the `Elements` [p.55] with a given tag name in the order in which they are encountered in a preorder traversal of the `Document` tree.

Parameters

`tagname` of type `DOMString` [p.17]

The name of the tag to match on. The special value "*" matches all tags.

Return Value

`NodeList` [p.44]	A new `NodeList` object containing all the matched `Elements` [p.55] .

No Exceptions

`getElementsByTagNameNS` introduced in **DOM Level 2**

Returns a `NodeList` [p.44] of all the `Elements` [p.55] with a given *local name* [p.102] and namespace URI in the order in which they are encountered in a preorder traversal of the `Document` tree.

Parameters

`namespaceURI` of type `DOMString` [p.17]

The *namespace URI* [p.103] of the elements to match on. The special value "*" matches all namespaces.

`localName` of type `DOMString`

The *local name* [p.102] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.44]	A new `NodeList` object containing all the matched `Elements` [p.55] .

No Exceptions

`importNode` introduced in **DOM Level 2**

Imports a node from another document to this document. The returned node has no parent; (`parentNode` is `null`). The source node is not altered or removed from the original document; this method creates a new copy of the source node.

For all nodes, importing a node creates a node object owned by the importing document, with attribute values identical to the source node's `nodeName` and `nodeType`, plus the attributes related to namespaces (`prefix`, `localName`, and `namespaceURI`). As in the `cloneNode` operation on a `Node` [p.35] , the source node is not altered.

Additional information is copied as appropriate to the `nodeType`, attempting to mirror the behavior expected if a fragment of XML or HTML source was copied from one document to another, recognizing that the two documents may have different DTDs in the XML case. The following list describes the specifics for each type of node.

ATTRIBUTE_NODE

The `ownerElement` attribute is set to `null` and the `specified` flag is set to `true` on the generated `Attr` [p.53] . The *descendants* [p.101] of the source `Attr` are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

Note that the `deep` parameter has no effect on `Attr` [p.53] nodes; they always carry their children with them when imported.

DOCUMENT_FRAGMENT_NODE

If the `deep` option was set to `true`, the *descendants* [p.101] of the source element are recursively imported and the resulting nodes reassembled to form the corresponding subtree. Otherwise, this simply generates an empty `DocumentFragment` [p.24] .

DOCUMENT_NODE

`Document` nodes cannot be imported.

DOCUMENT_TYPE_NODE

`DocumentType` [p.65] nodes cannot be imported.

ELEMENT_NODE

Specified attribute nodes of the source element are imported, and the generated `Attr` [p.53] nodes are attached to the generated `Element` [p.55] . Default attributes are *not* copied, though if the document being imported into defines default attributes for this element name, those are assigned. If the `importNode` `deep` parameter was set to `true`, the *descendants* [p.101] of the source element are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_NODE

`Entity` [p.67] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.65] is readonly. Ability to add these imported nodes to a `DocumentType` will be considered for addition to a future release of the DOM.

On import, the `publicId`, `systemId`, and `notationName` attributes are copied. If a `deep` import is requested, the *descendants* [p.101] of the the source `Entity` [p.67] are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_REFERENCE_NODE

Only the `EntityReference` [p.68] itself is copied, even if a `deep` import is requested, since the source and destination documents might have defined the entity differently. If the document being imported into provides a definition for this entity name, its value is assigned.

NOTATION_NODE

`Notation` [p.66] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.65] is readonly. Ability to add these imported nodes to a `DocumentType` will be considered for addition to a future release of the DOM.

On import, the `publicId` and `systemId` attributes are copied.

Note that the `deep` parameter has no effect on `Notation` [p.66] nodes since they never have any children.

PROCESSING_INSTRUCTION_NODE

The imported node copies its `target` and `data` values from those of the source node.

TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE

These three types of nodes inheriting from `CharacterData` [p.49] copy their `data` and `length` attributes from those of the source node.

Parameters

`importedNode` of type `Node` [p.35]

The node to import.

`deep` of type `boolean`

If `true`, recursively import the subtree under the specified node; if `false`, import only the node itself, as explained above. This has no effect on `Attr` [p.53] , `EntityReference` [p.68] , and `Notation` [p.66] nodes.

Return Value

`Node` [p.35]	The imported node that belongs to this `Document`.

Exceptions

`DOMException` [p.20]	NOT_SUPPORTED_ERR: Raised if the type of node being imported is not supported.

Interface *Node*

The `Node` interface is the primary datatype for the entire Document Object Model. It represents a single node in the document tree. While all objects implementing the `Node` interface expose methods for dealing with children, not all objects implementing the `Node` interface may have children. For example, `Text` [p.63] nodes may not have children, and adding children to such nodes results in a `DOMException` [p.20] being raised.

The attributes `nodeName`, `nodeValue` and `attributes` are included as a mechanism to get at node information without casting down to the specific derived interface. In cases where there is no obvious mapping of these attributes for a specific `nodeType` (e.g., `nodeValue` for an `Element` [p.55] or `attributes` for a `Comment` [p.64]), this returns `null`. Note that the specialized interfaces may contain additional and more convenient mechanisms to get and set the relevant information.

IDL Definition

```
interface Node {

  // NodeType
  const unsigned short      ELEMENT_NODE                   = 1;
  const unsigned short      ATTRIBUTE_NODE                 = 2;
  const unsigned short      TEXT_NODE                      = 3;
  const unsigned short      CDATA_SECTION_NODE             = 4;
  const unsigned short      ENTITY_REFERENCE_NODE          = 5;
  const unsigned short      ENTITY_NODE                    = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE    = 7;
  const unsigned short      COMMENT_NODE                   = 8;
```

```
  const unsigned short      DOCUMENT_NODE                  = 9;
  const unsigned short      DOCUMENT_TYPE_NODE             = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE         = 11;
  const unsigned short      NOTATION_NODE                  = 12;

  readonly attribute DOMString        nodeName;
           attribute DOMString        nodeValue;
                                        // raises(DOMException) on setting
                                        // raises(DOMException) on retrieval

  readonly attribute unsigned short   nodeType;
  readonly attribute Node             parentNode;
  readonly attribute NodeList         childNodes;
  readonly attribute Node             firstChild;
  readonly attribute Node             lastChild;
  readonly attribute Node             previousSibling;
  readonly attribute Node             nextSibling;
  readonly attribute NamedNodeMap     attributes;
  // Modified in DOM Level 2:
  readonly attribute Document         ownerDocument;
  Node               insertBefore(in Node newChild,
                                  in Node refChild)
                                        raises(DOMException);
  Node               replaceChild(in Node newChild,
                                  in Node oldChild)
                                        raises(DOMException);
  Node               removeChild(in Node oldChild)
                                        raises(DOMException);
  Node               appendChild(in Node newChild)
                                        raises(DOMException);
  boolean            hasChildNodes();
  Node               cloneNode(in boolean deep);
  // Modified in DOM Level 2:
  void               normalize();
  // Introduced in DOM Level 2:
  boolean            isSupported(in DOMString feature,
                                 in DOMString version);
  // Introduced in DOM Level 2:
  readonly attribute DOMString        namespaceURI;
  // Introduced in DOM Level 2:
           attribute DOMString        prefix;
                                        // raises(DOMException) on setting

  // Introduced in DOM Level 2:
  readonly attribute DOMString        localName;
  // Introduced in DOM Level 2:
  boolean            hasAttributes();
};
```

Definition group *NodeType*

An integer indicating which type of node this is.

Note: Numeric codes up to 200 are reserved to W3C for possible future use.

Defined Constants

ATTRIBUTE_NODE
: The node is an Attr [p.53] .

CDATA_SECTION_NODE
: The node is a CDATASection [p.64] .

COMMENT_NODE
: The node is a Comment [p.64] .

DOCUMENT_FRAGMENT_NODE
: The node is a DocumentFragment [p.24] .

DOCUMENT_NODE
: The node is a Document [p.25] .

DOCUMENT_TYPE_NODE
: The node is a DocumentType [p.65] .

ELEMENT_NODE
: The node is an Element [p.55] .

ENTITY_NODE
: The node is an Entity [p.67] .

ENTITY_REFERENCE_NODE
: The node is an EntityReference [p.68] .

NOTATION_NODE
: The node is a Notation [p.66] .

PROCESSING_INSTRUCTION_NODE
: The node is a ProcessingInstruction [p.68] .

TEXT_NODE
: The node is a Text [p.63] node.

The values of nodeName, nodeValue, and attributes vary according to the node type as follows:

	nodeName	nodeValue	attributes
Attr	name of attribute	value of attribute	null
CDATASection	#cdata-section	content of the CDATA Section	null
Comment	#comment	content of the comment	null
Document	#document	null	null
DocumentFragment	#document-fragment	null	null
DocumentType	document type name	null	null
Element	tag name	null	NamedNodeMap
Entity	entity name	null	null
EntityReference	name of entity referenced	null	null
Notation	notation name	null	null
ProcessingInstruction	target	entire content excluding the target	null
Text	#text	content of the text node	null

Attributes

`attributes` of type `NamedNodeMap` [p.45] , readonly
A `NamedNodeMap` [p.45] containing the attributes of this node (if it is an `Element` [p.55]) or `null` otherwise.

`childNodes` of type `NodeList` [p.44] , readonly
A `NodeList` [p.44] that contains all children of this node. If there are no children, this is a `NodeList` containing no nodes.

`firstChild` of type `Node` [p.35] , readonly
The first child of this node. If there is no such node, this returns `null`.

`lastChild` of type `Node` [p.35] , readonly
The last child of this node. If there is no such node, this returns `null`.

`localName` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
Returns the local part of the *qualified name* [p.103] of this node.
For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` [p.25] interface, this is always `null`.

`namespaceURI` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
The *namespace URI* [p.103] of this node, or `null` if it is unspecified.
This is not a computed value that is the result of a namespace lookup based on an examination of the namespace declarations in scope. It is merely the namespace URI given at creation time.
For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` [p.25] interface, this is always `null`.

Note: Per the *Namespaces in XML* Specification [Namespaces] an attribute does not inherit its namespace from the element it is attached to. If an attribute is not explicitly given a namespace, it simply has no namespace.

`nextSibling` of type `Node` [p.35] , readonly
The node immediately following this node. If there is no such node, this returns `null`.

`nodeName` of type `DOMString` [p.17] , readonly
The name of this node, depending on its type; see the table above.

`nodeType` of type `unsigned short`, readonly
A code representing the type of the underlying object, as defined above.

`nodeValue` of type `DOMString` [p.17]
The value of this node, depending on its type; see the table above. When it is defined to be `null`, setting it has no effect.
Exceptions on setting

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.20]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.17] variable on the implementation platform.

`ownerDocument` of type `Document` [p.25] , readonly, modified in **DOM Level 2**
The `Document` [p.25] object associated with this node. This is also the `Document` object used to create new nodes. When this node is a `Document` or a `DocumentType` [p.65] which is not used with any `Document` yet, this is `null`.

`parentNode` of type `Node` [p.35] , readonly
The *parent* [p.103] of this node. All nodes, except `Attr` [p.53] , `Document` [p.25] , `DocumentFragment` [p.24] , `Entity` [p.67] , and `Notation` [p.66] may have a parent. However, if a node has just been created and not yet added to the tree, or if it has been removed from the tree, this is `null`.

`prefix` of type `DOMString` [p.17] , introduced in **DOM Level 2**

The *namespace prefix* [p.103] of this node, or `null` if it is unspecified.
Note that setting this attribute, when permitted, changes the `nodeName` attribute, which holds the *qualified name* [p.103] , as well as the `tagName` and `name` attributes of the `Element` [p.55] and `Attr` [p.53] interfaces, when applicable.
Note also that changing the prefix of an attribute that is known to have a default value, does not make a new attribute with the default value and the original prefix appear, since the `namespaceURI` and `localName` do not change.
For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` [p.25] interface, this is always `null`.

Exceptions on setting

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified prefix contains an illegal character. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NAMESPACE_ERR: Raised if the specified `prefix` is malformed, if the `namespaceURI` of this node is `null`, if the specified prefix is "xml" and the `namespaceURI` of this node is different from "http://www.w3.org/XML/1998/namespace", if this node is an attribute and the specified prefix is "xmlns" and the `namespaceURI` of this node is different from "http://www.w3.org/2000/xmlns/", or if this node is an attribute and the `qualifiedName` of this node is "xmlns" [Namespaces].

`previousSibling` of type `Node` [p.35] , readonly

The node immediately preceding this node. If there is no such node, this returns `null`.

Methods

`appendChild`

Adds the node `newChild` to the end of the list of children of this node. If the `newChild` is already in the tree, it is first removed.

Parameters

`newChild` of type `Node` [p.35]

The node to add.
If it is a `DocumentFragment` [p.24] object, the entire contents of the document fragment are moved into the child list of this node

Return Value

`Node` [p.35]	The node added.

Exceptions

`DOMException` [p.20]	HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to append is one of this node's *ancestors* [p.101] . WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

`cloneNode`

Returns a duplicate of this node, i.e., serves as a generic copy constructor for nodes. The duplicate node has no parent; (`parentNode` is `null`.).
Cloning an `Element` [p.55] copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but this method does not copy any text it contains unless it is a deep clone, since the text is contained in a child `Text` [p.63] node. Cloning an `Attribute` directly, as opposed to be cloned as part of an `Element` cloning operation, returns a specified attribute (`specified` is `true`). Cloning any other type of node simply returns a copy of this node.
Note that cloning an immutable subtree results in a mutable copy, but the children of an `EntityReference` [p.68] clone are *readonly* [p.103] . In addition, clones of unspecified `Attr` [p.53] nodes are specified. And, cloning `Document` [p.25] , `DocumentType` [p.65] , `Entity` [p.67] , and `Notation` [p.66] nodes is implementation dependent.

Parameters

`deep` of type `boolean`
If `true`, recursively clone the subtree under the specified node; if `false`, clone only the node itself (and its attributes, if it is an `Element` [p.55]).

Return Value

`Node` [p.35]	The duplicate node.

No Exceptions

`hasAttributes` introduced in **DOM Level 2**

Returns whether this node (if it is an element) has any attributes.

Return Value

`boolean`	`true` if this node has any attributes, `false` otherwise.

No Parameters
No Exceptions

`hasChildNodes`

Returns whether this node has any children.

Return Value

`boolean`	`true` if this node has any children, `false` otherwise.

No Parameters
No Exceptions

`insertBefore`

Inserts the node `newChild` before the existing child node `refChild`. If `refChild` is `null`, insert `newChild` at the end of the list of children.
If `newChild` is a `DocumentFragment` [p.24] object, all of its children are inserted, in the same order, before `refChild`. If the `newChild` is already in the tree, it is first removed.

Parameters

`newChild` of type `Node` [p.35]
The node to insert.

`refChild` of type `Node`
The reference node, i.e., the node before which the new node must be inserted.

Return Value

`Node` [p.35]	The node being inserted.

Exceptions

`DOMException` [p.20]	HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to insert is one of this node's *ancestors* [p.101] . WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly or if the parent of the node being inserted is readonly. NOT_FOUND_ERR: Raised if `refChild` is not a child of this node.

`isSupported` introduced in **DOM Level 2**

Tests whether the DOM implementation implements a specific feature and that feature is supported by this node.

Parameters

`feature` of type `DOMString` [p.17]

The name of the feature to test. This is the same name which can be passed to the method `hasFeature` on `DOMImplementation` [p.22] .

`version` of type `DOMString`

This is the version number of the feature to test. In Level 2, version 1, this is the string "2.0". If the version is not specified, supporting any version of the feature will cause the method to return `true`.

Return Value

`boolean`	Returns `true` if the specified feature is supported on this node, `false` otherwise.

No Exceptions

`normalize` modified in **DOM Level 2**

Puts all `Text` [p.63] nodes in the full depth of the sub-tree underneath this `Node`, including attribute nodes, into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates `Text` nodes, i.e., there are neither adjacent `Text` nodes nor empty `Text` nodes. This can be used to ensure that the DOM view of a document is the same as if it were saved and re-loaded, and is useful when operations (such as XPointer [XPointer] lookups) that depend on a particular document tree structure are to be used.

Note: In cases where the document contains `CDATASections` [p.64] , the normalize operation alone may not be sufficient, since XPointers do not differentiate between `Text` [p.63] nodes and `CDATASection` [p.64] nodes.

No Parameters
No Return Value
No Exceptions

`removeChild`

Removes the child node indicated by `oldChild` from the list of children, and returns it.

Parameters

`oldChild` of type `Node` [p.35]

The node being removed.

Return Value

`Node` [p.35]	The node removed.

Exceptions

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if `oldChild` is not a child of this node.

`replaceChild`

Replaces the child node `oldChild` with `newChild` in the list of children, and returns the `oldChild` node.
If `newChild` is a `DocumentFragment` [p.24] object, `oldChild` is replaced by all of the `DocumentFragment` children, which are inserted in the same order. If the `newChild` is already in the tree, it is first removed.

Parameters

`newChild` of type `Node` [p.35]
The new node to put in the child list.

`oldChild` of type `Node`
The node being replaced in the list.

Return Value

`Node` [p.35]	The node replaced.

Exceptions

`DOMException` [p.20]	HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to put in is one of this node's *ancestors* [p.101] . WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node or the parent of the new node is readonly. NOT_FOUND_ERR: Raised if `oldChild` is not a child of this node.

Interface *NodeList*

The `NodeList` interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented. `NodeList` objects in the DOM are *live* [p.16] .

The items in the `NodeList` are accessible via an integral index, starting from 0.

IDL Definition

```
interface NodeList {
  Node               item(in unsigned long index);
  readonly attribute unsigned long    length;
};
```

Attributes

`length` of type `unsigned long`, readonly

The number of nodes in the list. The range of valid child node indices is 0 to `length-1` inclusive.

Methods

`item`

Returns the `index`th item in the collection. If `index` is greater than or equal to the number of nodes in the list, this returns `null`.

Parameters

`index` of type `unsigned long`

Index into the collection.

Return Value

`Node` [p.35]	The node at the `index`th position in the `NodeList`, or `null` if that is not a valid index.

No Exceptions

Interface *NamedNodeMap*

Objects implementing the `NamedNodeMap` interface are used to represent collections of nodes that can be accessed by name. Note that `NamedNodeMap` does not inherit from `NodeList` [p.44] ; `NamedNodeMaps` are not maintained in any particular order. Objects contained in an object implementing `NamedNodeMap` may also be accessed by an ordinal index, but this is simply to allow convenient enumeration of the contents of a `NamedNodeMap`, and does not imply that the DOM specifies an order to these Nodes.

`NamedNodeMap` objects in the DOM are *live* [p.16] .

IDL Definition

```
interface NamedNodeMap {
  Node               getNamedItem(in DOMString name);
  Node               setNamedItem(in Node arg)
                                        raises(DOMException);
  Node               removeNamedItem(in DOMString name)
```

```
                                        raises(DOMException);
  Node               item(in unsigned long index);
  readonly attribute unsigned long    length;
  // Introduced in DOM Level 2:
  Node               getNamedItemNS(in DOMString namespaceURI,
                                    in DOMString localName);
  // Introduced in DOM Level 2:
  Node               setNamedItemNS(in Node arg)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Node               removeNamedItemNS(in DOMString namespaceURI,
                                       in DOMString localName)
                                        raises(DOMException);
};
```

Attributes

`length` of type `unsigned long`, readonly
The number of nodes in this map. The range of valid child node indices is `0` to `length-1` inclusive.

Methods

`getNamedItem`
Retrieves a node specified by name.
Parameters
`name` of type `DOMString` [p.17]
The `nodeName` of a node to retrieve.

Return Value

`Node` [p.35]	A `Node` (of any type) with the specified `nodeName`, or `null` if it does not identify any node in this map.

No Exceptions

`getNamedItemNS` introduced in **DOM Level 2**
Retrieves a node specified by local name and namespace URI. HTML-only DOM implementations do not need to implement this method.
Parameters
`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the node to retrieve.

`localName` of type `DOMString`
The *local name* [p.102] of the node to retrieve.

Return Value

`Node` [p.35]	A `Node` (of any type) with the specified local name and namespace URI, or `null` if they do not identify any node in this map.

No Exceptions

`item`

Returns the `index`th item in the map. If `index` is greater than or equal to the number of nodes in this map, this returns `null`.

Parameters

`index` of type `unsigned long`

Index into this map.

Return Value

`Node` [p.35]	The node at the `index`th position in the map, or `null` if that is not a valid index.

No Exceptions

`removeNamedItem`

Removes a node specified by name. When this map contains the attributes attached to an element, if the removed attribute is known to have a default value, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.

Parameters

`name` of type `DOMString` [p.17]

The `nodeName` of the node to remove.

Return Value

`Node` [p.35]	The node removed from this map if a node with such a name exists.

Exceptions

`DOMException` [p.20]	NOT_FOUND_ERR: Raised if there is no node named `name` in this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly.

`removeNamedItemNS` introduced in **DOM Level 2**

Removes a node specified by local name and namespace URI. A removed attribute may be known to have a default value when this map contains the attributes attached to an element, as returned by the attributes attribute of the `Node` [p.35] interface. If so, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.

HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the node to remove.

`localName` of type `DOMString`
The *local name* [p.102] of the node to remove.

Return Value

`Node` [p.35]	The node removed from this map if a node with such a local name and namespace URI exists.

Exceptions

`DOMException` [p.20]	NOT_FOUND_ERR: Raised if there is no node with the specified `namespaceURI` and `localName` in this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly.

`setNamedItem`
Adds a node using its `nodeName` attribute. If a node with that name is already present in this map, it is replaced by the new one.
As the `nodeName` attribute is used to derive the name which the node must be stored under, multiple nodes of certain types (those that have a "special" string value) cannot be stored as the names would clash. This is seen as preferable to allowing nodes to be aliased.
Parameters
`arg` of type `Node` [p.35]
A node to store in this map. The node will later be accessible using the value of its `nodeName` attribute.

Return Value

`Node` [p.35]	If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.20]	WRONG_DOCUMENT_ERR: Raised if `arg` was created from a different document than the one that created this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly. INUSE_ATTRIBUTE_ERR: Raised if `arg` is an `Attr` [p.53] that is already an attribute of another `Element` [p.55] object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

`setNamedItemNS` introduced in **DOM Level 2**

Adds a node using its `namespaceURI` and `localName`. If a node with that namespace URI and that local name is already present in this map, it is replaced by the new one. HTML-only DOM implementations do not need to implement this method.

Parameters

`arg` of type `Node` [p.35]

A node to store in this map. The node will later be accessible using the value of its `namespaceURI` and `localName` attributes.

Return Value

`Node` [p.35]	If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.20]	WRONG_DOCUMENT_ERR: Raised if `arg` was created from a different document than the one that created this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly. INUSE_ATTRIBUTE_ERR: Raised if `arg` is an `Attr` [p.53] that is already an attribute of another `Element` [p.55] object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

Interface *CharacterData*

The `CharacterData` interface extends Node with a set of attributes and methods for accessing character data in the DOM. For clarity this set is defined here rather than on each object that uses these attributes and methods. No DOM objects correspond directly to `CharacterData`, though `Text` [p.63] and others do inherit the interface from it. All `offsets` in this interface start from `0`.

As explained in the `DOMString` [p.17] interface, text strings in the DOM are represented in UTF-16, i.e. as a sequence of 16-bit units. In the following, the term *16-bit units* [p.101] is used whenever necessary to indicate that indexing on CharacterData is done in 16-bit units.

IDL Definition

```
interface CharacterData : Node {
           attribute DOMString        data;
                                        // raises(DOMException) on setting
                                        // raises(DOMException) on retrieval

  readonly attribute unsigned long    length;
  DOMString          substringData(in unsigned long offset,
                                   in unsigned long count)
                                        raises(DOMException);
  void               appendData(in DOMString arg)
                                        raises(DOMException);
  void               insertData(in unsigned long offset,
                                in DOMString arg)
                                        raises(DOMException);
  void               deleteData(in unsigned long offset,
                                in unsigned long count)
                                        raises(DOMException);
  void               replaceData(in unsigned long offset,
                                 in unsigned long count,
                                 in DOMString arg)
                                        raises(DOMException);
};
```

Attributes

`data` of type `DOMString` [p.17]

The character data of the node that implements this interface. The DOM implementation may not put arbitrary limits on the amount of data that may be stored in a `CharacterData` node. However, implementation limits may mean that the entirety of a node's data may not fit into a single `DOMString` [p.17] . In such cases, the user may call `substringData` to retrieve the data in appropriately sized pieces.

Exceptions on setting

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.20]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.17] variable on the implementation platform.

`length` of type `unsigned long`, readonly

The number of *16-bit units* [p.101] that are available through `data` and the `substringData` method below. This may have the value zero, i.e., `CharacterData` nodes may be empty.

Methods

`appendData`

Append the string to the end of the character data of the node. Upon success, `data` provides access to the concatenation of `data` and the `DOMString` [p.17] specified.

Parameters

`arg` of type `DOMString` [p.17]

The `DOMString` to append.

Exceptions

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`deleteData`

Remove a range of *16-bit units* [p.101] from the node. Upon success, `data` and `length` reflect the change.

Parameters

`offset` of type `unsigned long`

The offset from which to start removing.

`count` of type `unsigned long`

The number of 16-bit units to delete. If the sum of `offset` and `count` exceeds `length` then all 16-bit units from `offset` to the end of the data are deleted.

Exceptions

`DOMException` [p.20]	INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`insertData`

Insert a string at the specified *16-bit unit* [p.101] offset.

Parameters

`offset` of type `unsigned long`

The character offset at which to insert.

`arg` of type `DOMString` [p.17]

The `DOMString` to insert.

Exceptions

`DOMException` [p.20]	INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`replaceData`

Replace the characters starting at the specified *16-bit unit* [p.101] offset with the specified string.

Parameters

`offset` of type `unsigned long`

The offset from which to start replacing.

`count` of type `unsigned long`

The number of 16-bit units to replace. If the sum of `offset` and `count` exceeds `length`, then all 16-bit units to the end of the data are replaced; (i.e., the effect is the same as a `remove` method call with the same range, followed by an `append` method invocation).

`arg` of type `DOMString` [p.17]

The `DOMString` with which the range must be replaced.

Exceptions

`DOMException` [p.20]	INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`substringData`

Extracts a range of data from the node.

Parameters

`offset` of type `unsigned long`

Start offset of substring to extract.

`count` of type `unsigned long`

The number of 16-bit units to extract.

Return Value

DOMString [p.17]	The specified substring. If the sum of offset and count exceeds the length, then all 16-bit units to the end of the data are returned.

Exceptions

DOMException [p.20]	INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in data, or if the specified count is negative. DOMSTRING_SIZE_ERR: Raised if the specified range of text does not fit into a DOMString [p.17] .

Interface *Attr*

The Attr interface represents an attribute in an Element [p.55] object. Typically the allowable values for the attribute are defined in a document type definition.

Attr objects inherit the Node [p.35] interface, but since they are not actually child nodes of the element they describe, the DOM does not consider them part of the document tree. Thus, the Node attributes parentNode, previousSibling, and nextSibling have a null value for Attr objects. The DOM takes the view that attributes are properties of elements rather than having a separate identity from the elements they are associated with; this should make it more efficient to implement such features as default attributes associated with all elements of a given type. Furthermore, Attr nodes may not be immediate children of a DocumentFragment [p.24] . However, they can be associated with Element [p.55] nodes contained within a DocumentFragment. In short, users and implementors of the DOM need to be aware that Attr nodes have some things in common with other objects inheriting the Node interface, but they also are quite distinct.

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 does not exist on this element in the structure model until it has been explicitly added. Note that the nodeValue attribute on the Attr instance can also be used to retrieve the string version of the attribute's value(s).

In XML, where the value of an attribute can contain entity references, the child nodes of the Attr node provide a representation in which entity references are not expanded. These child nodes may be either Text [p.63] or EntityReference [p.68] nodes. Because the DOM Core is not aware of attribute types, it treats all attribute values as simple strings, even if the DTD or schema declares them as having *tokenized* [p.103] types.

IDL Definition

```
interface Attr : Node {
  readonly attribute DOMString        name;
  readonly attribute boolean          specified;
           attribute DOMString        value;
                                        // raises(DOMException) on setting

  // Introduced in DOM Level 2:
  readonly attribute Element          ownerElement;
};
```

Attributes

`name` of type `DOMString` [p.17] , readonly
Returns the name of this attribute.

`ownerElement` of type `Element` [p.55] , readonly, introduced in **DOM Level 2**
The `Element` [p.55] node this attribute is attached to or `null` if this attribute is not in use.

`specified` of type `boolean`, readonly
If this attribute was explicitly given a value in the original document, this is `true`; otherwise, it is `false`. Note that the implementation is in charge of this attribute, not the user. If the user changes the value of the attribute (even if it ends up having the same value as the default value) then the `specified` flag is automatically flipped to `true`. To re-specify the attribute as the default value from the DTD, the user must delete the attribute. The implementation will then make a new attribute available with `specified` set to `false` and the default value (if one exists).
In summary:

- If the attribute has an assigned value in the document then `specified` is `true`, and the value is the assigned value.
- If the attribute has no assigned value in the document and has a default value in the DTD, then `specified` is `false`, and the value is the default value in the DTD.
- If the attribute has no assigned value in the document and has a value of #IMPLIED in the DTD, then the attribute does not appear in the structure model of the document.
- If the `ownerElement` attribute is `null` (i.e. because it was just created or was set to `null` by the various removal and cloning operations) `specified` is `true`.

`value` of type `DOMString` [p.17]
On retrieval, the value of the attribute is returned as a string. Character and general entity references are replaced with their values. See also the method `getAttribute` on the `Element` [p.55] interface.
On setting, this creates a `Text` [p.63] node with the unparsed contents of the string. I.e. any characters that an XML processor would recognize as markup are instead treated as literal text. See also the method `setAttribute` on the `Element` [p.55] interface.
Exceptions on setting

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Interface *Element*

The `Element` interface represents an *element* [p.102] in an HTML or XML document. Elements may have attributes associated with them; since the `Element` interface inherits from `Node` [p.35] , the generic `Node` interface attribute `attributes` may be used to retrieve the set of all attributes for an element. There are methods on the `Element` interface to retrieve either an `Attr` [p.53] object by name or an attribute value by name. In XML, where an attribute value may contain entity references, an `Attr` object should be retrieved to examine the possibly fairly complex sub-tree representing the attribute value. On the other hand, in HTML, where all attributes have simple string values, methods to directly access an attribute value can safely be used as a *convenience* [p.101] .

Note: In DOM Level 2, the method `normalize` is inherited from the `Node` [p.35] interface where it was moved.

IDL Definition

```
interface Element : Node {
  readonly attribute DOMString        tagName;
  DOMString          getAttribute(in DOMString name);
  void               setAttribute(in DOMString name, 
                                  in DOMString value)
                                        raises(DOMException);
  void               removeAttribute(in DOMString name)
                                        raises(DOMException);
  Attr               getAttributeNode(in DOMString name);
  Attr               setAttributeNode(in Attr newAttr)
                                        raises(DOMException);
  Attr               removeAttributeNode(in Attr oldAttr)
                                        raises(DOMException);
  NodeList           getElementsByTagName(in DOMString name);
  // Introduced in DOM Level 2:
  DOMString          getAttributeNS(in DOMString namespaceURI, 
                                    in DOMString localName);
  // Introduced in DOM Level 2:
  void               setAttributeNS(in DOMString namespaceURI, 
                                    in DOMString qualifiedName, 
                                    in DOMString value)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  void               removeAttributeNS(in DOMString namespaceURI, 
                                       in DOMString localName)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Attr               getAttributeNodeNS(in DOMString namespaceURI, 
                                        in DOMString localName);
  // Introduced in DOM Level 2:
  Attr               setAttributeNodeNS(in Attr newAttr)
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  NodeList           getElementsByTagNameNS(in DOMString namespaceURI, 
```

```
                                              in DOMString localName);
  // Introduced in DOM Level 2:
  boolean            hasAttribute(in DOMString name);
  // Introduced in DOM Level 2:
  boolean            hasAttributeNS(in DOMString namespaceURI,
                                    in DOMString localName);
};
```

Attributes

`tagName` of type `DOMString` [p.17] , readonly

The name of the element. For example, in:

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

`tagName` has the value `"elementExample"`. Note that this is case-preserving in XML, as are all of the operations of the DOM. The HTML DOM returns the `tagName` of an HTML element in the canonical uppercase form, regardless of the case in the source HTML document.

Methods

`getAttribute`

Retrieves an attribute value by name.

Parameters

`name` of type `DOMString` [p.17]

The name of the attribute to retrieve.

Return Value

`DOMString` [p.17]	The `Attr` [p.53] value as a string, or the empty string if that attribute does not have a specified or default value.

No Exceptions

`getAttributeNS` introduced in **DOM Level 2**

Retrieves an attribute value by local name and namespace URI. HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]

The *namespace URI* [p.103] of the attribute to retrieve.

`localName` of type `DOMString`

The *local name* [p.102] of the attribute to retrieve.

Return Value

`DOMString` [p.17]	The `Attr` [p.53] value as a string, or the empty string if that attribute does not have a specified or default value.

No Exceptions

`getAttributeNode`

Retrieves an attribute node by name.
To retrieve an attribute node by qualified name and namespace URI, use the `getAttributeNodeNS` method.

Parameters

`name` of type `DOMString` [p.17]
The name (`nodeName`) of the attribute to retrieve.

Return Value

`Attr` [p.53]	The `Attr` node with the specified name (`nodeName`) or `null` if there is no such attribute.

No Exceptions

`getAttributeNodeNS` introduced in **DOM Level 2**

Retrieves an `Attr` [p.53] node by local name and namespace URI. HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the attribute to retrieve.

`localName` of type `DOMString`
The *local name* [p.102] of the attribute to retrieve.

Return Value

`Attr` [p.53]	The `Attr` node with the specified attribute local name and namespace URI or `null` if there is no such attribute.

No Exceptions

`getElementsByTagName`

Returns a `NodeList` [p.44] of all *descendant* [p.101] `Elements` with a given tag name, in the order in which they are encountered in a preorder traversal of this `Element` tree.

Parameters

`name` of type `DOMString` [p.17]
The name of the tag to match on. The special value "*" matches all tags.

Return Value

`NodeList` [p.44]	A list of matching `Element` nodes.

No Exceptions

`getElementsByTagNameNS` introduced in **DOM Level 2**

Returns a `NodeList` [p.44] of all the *descendant* [p.101] `Elements` with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of this `Element` tree.
HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the elements to match on. The special value "*" matches all namespaces.

`localName` of type `DOMString`
The *local name* [p.102] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.44]	A new `NodeList` object containing all the matched `Elements`.

No Exceptions

`hasAttribute` introduced in **DOM Level 2**

Returns `true` when an attribute with a given name is specified on this element or has a default value, `false` otherwise.

Parameters

`name` of type `DOMString` [p.17]
The name of the attribute to look for.

Return Value

`boolean`	`true` if an attribute with the given name is specified on this element or has a default value, `false` otherwise.

No Exceptions

`hasAttributeNS` introduced in **DOM Level 2**

Returns `true` when an attribute with a given local name and namespace URI is specified on this element or has a default value, `false` otherwise. HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the attribute to look for.

`localName` of type `DOMString`
The *local name* [p.102] of the attribute to look for.

Return Value

`boolean`	`true` if an attribute with the given local name and namespace URI is specified or has a default value on this element, `false` otherwise.

No Exceptions

`removeAttribute`
Removes an attribute by name. If the removed attribute is known to have a default value, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.
To remove an attribute by local name and namespace URI, use the `removeAttributeNS` method.
Parameters
`name` of type `DOMString` [p.17]
The name of the attribute to remove.

Exceptions

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`removeAttributeNS` introduced in **DOM Level 2**
Removes an attribute by local name and namespace URI. If the removed attribute has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix.
HTML-only DOM implementations do not need to implement this method.
Parameters
`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the attribute to remove.

`localName` of type `DOMString`
The *local name* [p.102] of the attribute to remove.

Exceptions

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`removeAttributeNode`

Removes the specified attribute node. If the removed `Attr` [p.53] has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix, when applicable.

Parameters

`oldAttr` of type `Attr` [p.53]

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

Return Value

`Attr` [p.53]	The `Attr` node that was removed.

Exceptions

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if `oldAttr` is not an attribute of the element.

`setAttribute`

Adds a new attribute. If an attribute with that name is already present in the element, its value is changed to be that of the value parameter. This value is a simple string; it is not parsed as it is being set. So any markup (such as syntax to be recognized as an entity reference) is treated as literal text, and needs to be appropriately escaped by the implementation when it is written out. In order to assign an attribute value that contains entity references, the user must create an `Attr` [p.53] node plus any `Text` [p.63] and `EntityReference` [p.68] nodes, build the appropriate subtree, and use `setAttributeNode` to assign it as the value of an attribute.

To set an attribute with a qualified name and namespace URI, use the `setAttributeNS` method.

Parameters

`name` of type `DOMString` [p.17]

The name of the attribute to create or alter.

`value` of type `DOMString`

Value to set in string form.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`setAttributeNS` introduced in **DOM Level 2**

Adds a new attribute. If an attribute with the same local name and namespace URI is already present on the element, its prefix is changed to be the prefix part of the `qualifiedName`, and its value is changed to be the `value` parameter. This value is a simple string; it is not parsed as it is being set. So any markup (such as syntax to be recognized as an entity reference) is treated as literal text, and needs to be appropriately escaped by the implementation when it is written out. In order to assign an attribute value that contains entity references, the user must create an `Attr` [p.53] node plus any `Text` [p.63] and `EntityReference` [p.68] nodes, build the appropriate subtree, and use `setAttributeNodeNS` or `setAttributeNode` to assign it as the value of an attribute.

HTML-only DOM implementations do not need to implement this method.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.103] of the attribute to create or alter.

`qualifiedName` of type `DOMString`
The *qualified name* [p.103] of the attribute to create or alter.

`value` of type `DOMString`
The value to set in string form.

Exceptions

`DOMException` [p.20]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from "http://www.w3.org/XML/1998/namespace", or if the `qualifiedName` is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/".

No Return Value

`setAttributeNode`

Adds a new attribute node. If an attribute with that name (`nodeName`) is already present in the element, it is replaced by the new one.
To add a new attribute node with a qualified name and namespace URI, use the `setAttributeNodeNS` method.

Parameters

`newAttr` of type `Attr` [p.53]
The `Attr` node to add to the attribute list.

Return Value

`Attr` [p.53]	If the `newAttr` attribute replaces an existing attribute, the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.20]	WRONG_DOCUMENT_ERR: Raised if `newAttr` was created from a different document than the one that created the element. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. INUSE_ATTRIBUTE_ERR: Raised if `newAttr` is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` [p.53] nodes to re-use them in other elements.

`setAttributeNodeNS` introduced in **DOM Level 2**

Adds a new attribute. If an attribute with that local name and that namespace URI is already present in the element, it is replaced by the new one.
HTML-only DOM implementations do not need to implement this method.

Parameters

`newAttr` of type `Attr` [p.53]
The `Attr` node to add to the attribute list.

Return Value

`Attr` [p.53]	If the `newAttr` attribute replaces an existing attribute with the same *local name* [p.102] and *namespace URI* [p.103] , the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.20]	WRONG_DOCUMENT_ERR: Raised if `newAttr` was created from a different document than the one that created the element. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. INUSE_ATTRIBUTE_ERR: Raised if `newAttr` is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` [p.53] nodes to re-use them in other elements.

Interface *Text*

The `Text` interface inherits from `CharacterData` [p.49] and represents the textual content (termed *character data* in XML) of an `Element` [p.55] or `Attr` [p.53] . If there is no markup inside an element's content, the text is contained in a single object implementing the `Text` interface that is the only child of the element. If there is markup, it is parsed into the *information items* [p.102] (elements, comments, etc.) and `Text` nodes that form the list of children of the element.

When a document is first made available via the DOM, there is only one `Text` node for each block of text. Users may create adjacent `Text` nodes that represent the contents of a given element without any intervening markup, but should be aware that there is no way to represent the separations between these nodes in XML or HTML, so they will not (in general) persist between DOM editing sessions. The `normalize()` method on `Node` [p.35] merges any such adjacent `Text` objects into a single node for each block of text.

IDL Definition

```
interface Text : CharacterData {
  Text               splitText(in unsigned long offset)
                                        raises(DOMException);
};
```

Methods

`splitText`

Breaks this node into two nodes at the specified `offset`, keeping both in the tree as *siblings* [p.103] . After being split, this node will contain all the content up to the `offset` point. A new node of the same type, which contains all the content at and after the `offset` point, is returned. If the original node had a parent node, the new node is inserted as the next *sibling* [p.103] of the original node. When the `offset` is equal to the length of this node, the new node has no data.

Parameters

`offset` of type `unsigned long`

The *16-bit unit* [p.101] offset at which to split, starting from `0`.

Return Value

`Text` [p.63]	The new node, of the same type as this node.

Exceptions

`DOMException` [p.20]	INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in `data`. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

Interface *Comment*

This interface inherits from `CharacterData` [p.49] and 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 : CharacterData {
};
```

1.3. Extended Interfaces

The interfaces defined here form part of the DOM Core specification, but objects that expose these interfaces will never be encountered in a DOM implementation that deals only with HTML. As such, HTML-only DOM implementations [DOM Level 2 HTML] do not need to have objects that implement these interfaces.

A DOM application can use the `hasFeature` method of the `DOMImplementation` [p.22] interface to determine whether they are supported or not. The feature string for all the interfaces listed in this section is "XML" and the version is "2.0".

Interface *CDATASection*

CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup. The only delimiter that is recognized in a CDATA section is the "]]>" string that ends the CDATA section. CDATA sections cannot be nested. Their primary purpose is for including material such as XML fragments, without needing to escape all the delimiters.

The `DOMString` [p.17] attribute of the `Text` [p.63] node holds the text that is contained by the CDATA section. Note that this *may* contain characters that need to be escaped outside of CDATA sections and that, depending on the character encoding ("charset") chosen for serialization, it may be impossible to write out some characters as part of a CDATA section.

The `CDATASection` interface inherits from the `CharacterData` [p.49] interface through the `Text` [p.63] interface. Adjacent `CDATASection` nodes are not merged by use of the `normalize` method of the `Node` [p.35] interface.

Note: Because no markup is recognized within a `CDATASection`, character numeric references cannot be used as an escape mechanism when serializing. Therefore, action needs to be taken when serializing a `CDATASection` with a character encoding where some of the contained characters cannot be represented. Failure to do so would not produce well-formed XML.
One potential solution in the serialization process is to end the CDATA section before the character, output the character using a character reference or entity reference, and open a new CDATA section for any further characters in the text node. Note, however, that some code conversion libraries at the time of writing do not return an error or exception when a character is missing from the encoding, making the task of ensuring that data is not corrupted on serialization more difficult.

IDL Definition

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

Interface *DocumentType*

Each `Document` [p.25] has a `doctype` attribute whose value is either `null` or a `DocumentType` object. The `DocumentType` interface in the DOM Core provides an interface to the list of entities that are defined for the document, and little else because the effect of namespaces and the various XML schema efforts on DTD representation are not clearly understood as of this writing.

The DOM Level 2 doesn't support editing `DocumentType` nodes.
IDL Definition

```
interface DocumentType : Node {
  readonly attribute DOMString        name;
  readonly attribute NamedNodeMap     entities;
  readonly attribute NamedNodeMap     notations;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        publicId;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        systemId;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        internalSubset;
};
```

Attributes

`entities` of type `NamedNodeMap` [p.45] , readonly
A `NamedNodeMap` [p.45] containing the general entities, both external and internal, declared in the DTD. Parameter entities are not contained. Duplicates are discarded. For example in:

```
<!DOCTYPE ex SYSTEM "ex.dtd" [
  <!ENTITY foo "foo">
  <!ENTITY bar "bar">
  <!ENTITY bar "bar2">
  <!ENTITY % baz "baz">
]>
<ex/>
```

the interface provides access to `foo` and the first declaration of `bar` but not the second declaration of `bar` or `baz`. Every node in this map also implements the `Entity` [p.67] interface.
The DOM Level 2 does not support editing entities, therefore `entities` cannot be altered in any way.

`internalSubset` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
The internal subset as a string.

Note: The actual content returned depends on how much information is available to the implementation. This may vary depending on various parameters, including the XML processor used to build the document.

`name` of type `DOMString` [p.17] , readonly
The name of DTD; i.e., the name immediately following the `DOCTYPE` keyword.

`notations` of type `NamedNodeMap` [p.45] , readonly
A `NamedNodeMap` [p.45] containing the notations declared in the DTD. Duplicates are discarded. Every node in this map also implements the `Notation` [p.66] interface.
The DOM Level 2 does not support editing notations, therefore `notations` cannot be altered in any way.

`publicId` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
The public identifier of the external subset.

`systemId` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
The system identifier of the external subset.

Interface *Notation*

This interface represents a notation declared in the DTD. A notation either declares, by name, the format of an unparsed entity (see *section 4.7* of the XML 1.0 specification [XML]), or is used for formal declaration of processing instruction targets (see *section 2.6* of the XML 1.0 specification [XML]). The `nodeName` attribute inherited from `Node` [p.35] is set to the declared name of the notation.

The DOM Level 1 does not support editing `Notation` nodes; they are therefore *readonly* [p.103] .

A `Notation` node does not have any parent.
IDL Definition

```
interface Notation : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
};
```

Attributes

`publicId` of type `DOMString` [p.17] , readonly
: The public identifier of this notation. If the public identifier was not specified, this is `null`.

`systemId` of type `DOMString` [p.17] , readonly
: The system identifier of this notation. If the system identifier was not specified, this is `null`.

Interface *Entity*

This interface represents an entity, either parsed or unparsed, in an XML document. Note that this models the entity itself *not* the entity declaration. `Entity` declaration modeling has been left for a later Level of the DOM specification.

The `nodeName` attribute that is inherited from `Node` [p.35] contains the name of the entity.

An XML processor may choose to completely expand entities before the structure model is passed to the DOM; in this case there will be no `EntityReference` [p.68] nodes in the document tree.

XML does not mandate that a non-validating XML processor read and process entity declarations made in the external subset or declared in external parameter entities. This means that parsed entities declared in the external subset need not be expanded by some classes of applications, and that the replacement value of the entity may not be available. When the replacement value is available, the corresponding `Entity` node's child list represents the structure of that replacement text. Otherwise, the child list is empty.

The DOM Level 2 does not support editing `Entity` nodes; if a user wants to make changes to the contents of an `Entity`, every related `EntityReference` [p.68] node has to be replaced in the structure model by a clone of the `Entity`'s contents, and then the desired changes must be made to each of those clones instead. `Entity` nodes and all their *descendants* [p.101] are *readonly* [p.103] .

An `Entity` node does not have any parent.

Note: If the entity contains an unbound *namespace prefix* [p.103] , the `namespaceURI` of the corresponding node in the `Entity` node subtree is `null`. The same is true for `EntityReference` [p.68] nodes that refer to this entity, when they are created using the `createEntityReference` method of the `Document` [p.25] interface. The DOM Level 2 does not support any mechanism to resolve namespace prefixes.

IDL Definition

```
interface Entity : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
  readonly attribute DOMString        notationName;
};
```

Attributes

`notationName` of type `DOMString` [p.17] , readonly
: For unparsed entities, the name of the notation for the entity. For parsed entities, this is `null`.

`publicId` of type `DOMString` [p.17] , readonly
: The public identifier associated with the entity, if specified. If the public identifier was not specified, this is `null`.

`systemId` of type `DOMString` [p.17] , readonly
: The system identifier associated with the entity, if specified. If the system identifier was not specified, this is `null`.

Interface *EntityReference*

`EntityReference` objects may be inserted into the structure model when an entity reference is in the source document, or when the user wishes to insert an entity reference. Note that character references and references to predefined entities are considered to be expanded by the HTML or XML processor so that characters are represented by their Unicode equivalent rather than by an entity reference. Moreover, the XML processor may completely expand references to entities while building the structure model, instead of providing `EntityReference` objects. If it does provide such objects, then for a given `EntityReference` node, it may be that there is no `Entity` [p.67] node representing the referenced entity. If such an `Entity` exists, then the subtree of the `EntityReference` node is in general a copy of the `Entity` node subtree. However, this may not be true when an entity contains an unbound *namespace prefix* [p.103] . In such a case, because the namespace prefix resolution depends on where the entity reference is, the *descendants* [p.101] of the `EntityReference` node may be bound to different *namespace URIs* [p.103] .

As for `Entity` [p.67] nodes, `EntityReference` nodes and all their *descendants* [p.101] are *readonly* [p.103] .

IDL Definition

```
interface EntityReference : Node {
};
```

Interface *ProcessingInstruction*

The `ProcessingInstruction` interface represents a "processing instruction", used in XML as a way to keep processor-specific information in the text of the document.

IDL Definition

```
interface ProcessingInstruction : Node {
  readonly attribute DOMString        target;
           attribute DOMString        data;
                                        // raises(DOMException) on setting

};
```

Attributes

`data` of type `DOMString` [p.17]

The content of this processing instruction. This is from the first non white space character after the target to the character immediately preceding the `?>`.

Exceptions on setting

`DOMException` [p.20]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

`target` of type `DOMString` [p.17] , readonly

The target of this processing instruction. XML defines this as being the first *token* [p.103] following the markup that begins the processing instruction.

Appendix A: Changes

Editors

Arnaud Le Hors, IBM
Philippe Le Hégaret, W3C

A.1: Changes between DOM Level 1 Core and DOM Level 2 Core

A.1.1: Changes to DOM Level 1 Core interfaces and exceptions

Interface `Attr` [p.53]
: The `Attr` [p.53] interface has one new attribute: `ownerElement`.

Interface `Document` [p.25]
: The `Document` [p.25] interface has five new methods: `importNode`, `createElementNS`, `createAttributeNS`, `getElementsByTagNameNS` and `getElementById`.

Interface `NamedNodeMap` [p.45]
: The `NamedNodeMap` [p.45] interface has three new methods: `getNamedItemNS`, `setNamedItemNS`, `removeNamedItemNS`.

Interface `Node` [p.35]
: The `Node` [p.35] interface has one new method: `supports`.
`normalize`, previously in the `Element` [p.55] interface, has been moved in the `Node` [p.35] interface.
The `Node` [p.35] interface has three new attributes: `namespaceURI`, `prefix` and `localName`.
The `ownerDocument` attribute was specified to be `null` when the node is a `Document` [p.25] . It now is also `null` when the node is a `DocumentType` [p.65] which is not used with any `Document` yet.

Interface `DocumentType` [p.65]
: The `DocumentType` [p.65] interface has three attributes: `publicId`, `systemId` and `internalSubset`.

Interface `DOMImplementation` [p.22]
: The `DOMImplementation` [p.22] interface has two new methods: `createDocumentType` and `createDocument`.

Interface `Element` [p.55]
: The `Element` [p.55] interface has eight new methods: `getAttributeNS`, `setAttributeNS`, `removeAttributeNS`, `getAttributeNodeNS`, `setAttributeNodeNS`, `getElementsByTagNameNS`, `hasAttribute` and `hasAttributeNS`.
The method `normalize` is now inherited from the `Node` [p.35] interface where it was moved.

Exception `DOMException` [p.20]
: The `DOMException` [p.20] has five new exception codes: `INVALID_STATE_ERR`, `SYNTAX_ERR`, `INVALID_MODIFICATION_ERR`, `NAMESPACE_ERR` and `INVALID_ACCESS_ERR`.

A.1.2: New features

A.1.2.1: New types

`DOMTimeStamp` [p.18]

The `DOMTimeStamp` [p.18] type was added to the Core module.

Appendix B: Accessing code point boundaries

Mark Davis, IBM
Lauren Wood, SoftQuad Software Inc.

B.1: Introduction

This appendix is an informative, not a normative, part of the Level 2 DOM specification.

Characters are represented in Unicode by numbers called *code points* (also called *scalar values*). These numbers can range from 0 up to 1,114,111 = $10FFFF_{16}$ (although some of these values are illegal). Each code point can be directly encoded with a 32-bit code unit. This encoding is termed UCS-4 (or UTF-32). The DOM specification, however, uses UTF-16, in which the most frequent characters (which have values less than $FFFF_{16}$) are represented by a single 16-bit code unit, while characters above $FFFF_{16}$ use a special pair of code units called a *surrogate pair*. For more information, see [Unicode] or the Unicode Web site.

While indexing by code points as opposed to code units is not common in programs, some specifications such as XPath (and therefore XSLT and XPointer) use code point indices. For interfacing with such formats it is recommended that the programming language provide string processing methods for converting code point indices to code unit indices and back. Some languages do not provide these functions natively; for these it is recommended that the native `String` type that is bound to `DOMString` [p.17] be extended to enable this conversion. An example of how such an API might look is supplied below.

Note: Since these methods are supplied as an illustrative example of the type of functionality that is required, the names of the methods, exceptions, and interface may differ from those given here.

B.2: Methods

Interface *StringExtend*

Extensions to a language's native String class or interface
IDL Definition

```
interface StringExtend {
  int                findOffset16(in int offset32)
                                        raises(StringIndexOutOfBoundsException);
  int                findOffset32(in int offset16)
                                        raises(StringIndexOutOfBoundsException);
};
```

Methods

`findOffset16`

Returns the UTF-16 offset that corresponds to a UTF-32 offset. Used for random access.

Note: You can always roundtrip from a UTF-32 offset to a UTF-16 offset and back. You can roundtrip from a UTF-16 offset to a UTF-32 offset and back if and only if the offset16 is not in the middle of a surrogate pair. Unmatched surrogates count as a single UTF-16 value.

Parameters

`offset32` of type `int`
: UTF-32 offset.

Return Value

`int`	UTF-16 offset

Exceptions

`StringIndexOutOfBoundsException`	if `offset32` is out of bounds.

`findOffset32`

Returns the UTF-32 offset corresponding to a UTF-16 offset. Used for random access. To find the UTF-32 length of a string, use:

```
len32 = findOffset32(source, source.length());
```

Note: If the UTF-16 offset is into the middle of a surrogate pair, then the UTF-32 offset of the *end* of the pair is returned; that is, the index of the char after the end of the pair. You can always roundtrip from a UTF-32 offset to a UTF-16 offset and back. You can roundtrip from a UTF-16 offset to a UTF-32 offset and back if and only if the offset16 is not in the middle of a surrogate pair. Unmatched surrogates count as a single UTF-16 value.

Parameters

`offset16` of type `int`
: UTF-16 offset

Return Value

`int`	UTF-32 offset

Exceptions

`StringIndexOutOfBoundsException`	if offset16 is out of bounds.

Appendix C: IDL Definitions

This appendix contains the complete OMG IDL [OMGIDL] for the Level 2 Document Object Model Core definitions.

The IDL files are also available as: http://www.w3.org/TR/2000/PR-DOM-Level-2-Core-20000927/idl.zip

dom.idl:

```
// File: dom.idl

#ifndef _DOM_IDL_
#define _DOM_IDL_

#pragma prefix "w3c.org"
module dom
{

  typedef sequence<unsigned short> DOMString;

  typedef   unsigned long long DOMTimeStamp;

  interface DocumentType;
  interface Document;
  interface NodeList;
  interface NamedNodeMap;
  interface Element;

  exception DOMException {
    unsigned short   code;
  };
  // ExceptionCode
  const unsigned short      INDEX_SIZE_ERR                 = 1;
  const unsigned short      DOMSTRING_SIZE_ERR             = 2;
  const unsigned short      HIERARCHY_REQUEST_ERR          = 3;
  const unsigned short      WRONG_DOCUMENT_ERR             = 4;
  const unsigned short      INVALID_CHARACTER_ERR          = 5;
  const unsigned short      NO_DATA_ALLOWED_ERR            = 6;
  const unsigned short      NO_MODIFICATION_ALLOWED_ERR    = 7;
  const unsigned short      NOT_FOUND_ERR                  = 8;
  const unsigned short      NOT_SUPPORTED_ERR              = 9;
  const unsigned short      INUSE_ATTRIBUTE_ERR            = 10;
  // Introduced in DOM Level 2:
  const unsigned short      INVALID_STATE_ERR              = 11;
  // Introduced in DOM Level 2:
  const unsigned short      SYNTAX_ERR                     = 12;
  // Introduced in DOM Level 2:
  const unsigned short      INVALID_MODIFICATION_ERR       = 13;
  // Introduced in DOM Level 2:
  const unsigned short      NAMESPACE_ERR                  = 14;
  // Introduced in DOM Level 2:
  const unsigned short      INVALID_ACCESS_ERR             = 15;
```

```
interface DOMImplementation {
  boolean            hasFeature(in DOMString feature,
                                in DOMString version);
  // Introduced in DOM Level 2:
  DocumentType       createDocumentType(in DOMString qualifiedName,
                                        in DOMString publicId,
                                        in DOMString systemId)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  Document           createDocument(in DOMString namespaceURI,
                                    in DOMString qualifiedName,
                                    in DocumentType doctype)
                                      raises(DOMException);
};

interface Node {

  // NodeType
  const unsigned short      ELEMENT_NODE                   = 1;
  const unsigned short      ATTRIBUTE_NODE                 = 2;
  const unsigned short      TEXT_NODE                      = 3;
  const unsigned short      CDATA_SECTION_NODE             = 4;
  const unsigned short      ENTITY_REFERENCE_NODE          = 5;
  const unsigned short      ENTITY_NODE                    = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE    = 7;
  const unsigned short      COMMENT_NODE                   = 8;
  const unsigned short      DOCUMENT_NODE                  = 9;
  const unsigned short      DOCUMENT_TYPE_NODE             = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE         = 11;
  const unsigned short      NOTATION_NODE                  = 12;

  readonly attribute DOMString        nodeName;
           attribute DOMString        nodeValue;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval

  readonly attribute unsigned short   nodeType;
  readonly attribute Node             parentNode;
  readonly attribute NodeList         childNodes;
  readonly attribute Node             firstChild;
  readonly attribute Node             lastChild;
  readonly attribute Node             previousSibling;
  readonly attribute Node             nextSibling;
  readonly attribute NamedNodeMap     attributes;
  // Modified in DOM Level 2:
  readonly attribute Document         ownerDocument;
  Node               insertBefore(in Node newChild,
                                  in Node refChild)
                                      raises(DOMException);
  Node               replaceChild(in Node newChild,
                                  in Node oldChild)
                                      raises(DOMException);
  Node               removeChild(in Node oldChild)
                                      raises(DOMException);
  Node               appendChild(in Node newChild)
                                      raises(DOMException);
  boolean            hasChildNodes();
```

```
  Node                cloneNode(in boolean deep);
  // Modified in DOM Level 2:
  void                normalize();
  // Introduced in DOM Level 2:
  boolean             isSupported(in DOMString feature,
                                  in DOMString version);
  // Introduced in DOM Level 2:
  readonly attribute DOMString        namespaceURI;
  // Introduced in DOM Level 2:
           attribute DOMString        prefix;
                                      // raises(DOMException) on setting

  // Introduced in DOM Level 2:
  readonly attribute DOMString        localName;
  // Introduced in DOM Level 2:
  boolean             hasAttributes();
};

interface NodeList {
  Node                item(in unsigned long index);
  readonly attribute unsigned long    length;
};

interface NamedNodeMap {
  Node                getNamedItem(in DOMString name);
  Node                setNamedItem(in Node arg)
                                      raises(DOMException);
  Node                removeNamedItem(in DOMString name)
                                      raises(DOMException);
  Node                item(in unsigned long index);
  readonly attribute unsigned long    length;
  // Introduced in DOM Level 2:
  Node                getNamedItemNS(in DOMString namespaceURI,
                                     in DOMString localName);
  // Introduced in DOM Level 2:
  Node                setNamedItemNS(in Node arg)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  Node                removeNamedItemNS(in DOMString namespaceURI,
                                       in DOMString localName)
                                      raises(DOMException);
};

interface CharacterData : Node {
           attribute DOMString        data;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval

  readonly attribute unsigned long    length;
  DOMString           substringData(in unsigned long offset,
                                    in unsigned long count)
                                      raises(DOMException);
  void                appendData(in DOMString arg)
                                      raises(DOMException);
  void                insertData(in unsigned long offset,
                                 in DOMString arg)
                                      raises(DOMException);
```

```
  void               deleteData(in unsigned long offset,
                                in unsigned long count)
                                      raises(DOMException);
  void               replaceData(in unsigned long offset,
                                 in unsigned long count,
                                 in DOMString arg)
                                      raises(DOMException);
};

interface Attr : Node {
  readonly attribute DOMString        name;
  readonly attribute boolean          specified;
           attribute DOMString        value;
                                      // raises(DOMException) on setting

  // Introduced in DOM Level 2:
  readonly attribute Element          ownerElement;
};

interface Element : Node {
  readonly attribute DOMString        tagName;
  DOMString          getAttribute(in DOMString name);
  void               setAttribute(in DOMString name,
                                  in DOMString value)
                                      raises(DOMException);
  void               removeAttribute(in DOMString name)
                                      raises(DOMException);
  Attr               getAttributeNode(in DOMString name);
  Attr               setAttributeNode(in Attr newAttr)
                                      raises(DOMException);
  Attr               removeAttributeNode(in Attr oldAttr)
                                      raises(DOMException);
  NodeList           getElementsByTagName(in DOMString name);
  // Introduced in DOM Level 2:
  DOMString          getAttributeNS(in DOMString namespaceURI,
                                    in DOMString localName);
  // Introduced in DOM Level 2:
  void               setAttributeNS(in DOMString namespaceURI,
                                    in DOMString qualifiedName,
                                    in DOMString value)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  void               removeAttributeNS(in DOMString namespaceURI,
                                       in DOMString localName)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  Attr               getAttributeNodeNS(in DOMString namespaceURI,
                                        in DOMString localName);
  // Introduced in DOM Level 2:
  Attr               setAttributeNodeNS(in Attr newAttr)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  NodeList           getElementsByTagNameNS(in DOMString namespaceURI,
                                            in DOMString localName);
  // Introduced in DOM Level 2:
  boolean            hasAttribute(in DOMString name);
  // Introduced in DOM Level 2:
```

```
  boolean             hasAttributeNS(in DOMString namespaceURI,
                                     in DOMString localName);
};

interface Text : CharacterData {
  Text                splitText(in unsigned long offset)
                                      raises(DOMException);
};

interface Comment : CharacterData {
};

interface CDATASection : Text {
};

interface DocumentType : Node {
  readonly attribute DOMString        name;
  readonly attribute NamedNodeMap     entities;
  readonly attribute NamedNodeMap     notations;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        publicId;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        systemId;
  // Introduced in DOM Level 2:
  readonly attribute DOMString        internalSubset;
};

interface Notation : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
};

interface Entity : Node {
  readonly attribute DOMString        publicId;
  readonly attribute DOMString        systemId;
  readonly attribute DOMString        notationName;
};

interface EntityReference : Node {
};

interface ProcessingInstruction : Node {
  readonly attribute DOMString        target;
           attribute DOMString        data;
                                      // raises(DOMException) on setting

};

interface DocumentFragment : Node {
};

interface Document : Node {
  readonly attribute DocumentType     doctype;
  readonly attribute DOMImplementation  implementation;
  readonly attribute Element          documentElement;
  Element             createElement(in DOMString tagName)
                                      raises(DOMException);
```

```
    DocumentFragment   createDocumentFragment();
    Text               createTextNode(in DOMString data);
    Comment            createComment(in DOMString data);
    CDATASection       createCDATASection(in DOMString data)
                                 raises(DOMException);
    ProcessingInstruction createProcessingInstruction(in DOMString target,
                                              in DOMString data)
                                 raises(DOMException);
    Attr               createAttribute(in DOMString name)
                                 raises(DOMException);
    EntityReference    createEntityReference(in DOMString name)
                                 raises(DOMException);
    NodeList           getElementsByTagName(in DOMString tagname);
    // Introduced in DOM Level 2:
    Node               importNode(in Node importedNode,
                                  in boolean deep)
                                 raises(DOMException);
    // Introduced in DOM Level 2:
    Element            createElementNS(in DOMString namespaceURI,
                                      in DOMString qualifiedName)
                                 raises(DOMException);
    // Introduced in DOM Level 2:
    Attr               createAttributeNS(in DOMString namespaceURI,
                                         in DOMString qualifiedName)
                                 raises(DOMException);
    // Introduced in DOM Level 2:
    NodeList           getElementsByTagNameNS(in DOMString namespaceURI,
                                              in DOMString localName);
    // Introduced in DOM Level 2:
    Element            getElementById(in DOMString elementId);
  };
};

#endif // _DOM_IDL_
```

Appendix D: Java Language Binding

This appendix contains the complete Java [Java] bindings for the Level 2 Document Object Model Core.

The Java files are also available as
http://www.w3.org/TR/2000/PR-DOM-Level-2-Core-20000927/java-binding.zip

org/w3c/dom/DOMException.java:

```
package org.w3c.dom;

public class DOMException extends RuntimeException {
    public DOMException(short code, String message) {
       super(message);
       this.code = code;
    }
    public short   code;
    // ExceptionCode
    public static final short INDEX_SIZE_ERR            = 1;
    public static final short DOMSTRING_SIZE_ERR        = 2;
    public static final short HIERARCHY_REQUEST_ERR     = 3;
    public static final short WRONG_DOCUMENT_ERR        = 4;
    public static final short INVALID_CHARACTER_ERR     = 5;
    public static final short NO_DATA_ALLOWED_ERR       = 6;
    public static final short NO_MODIFICATION_ALLOWED_ERR = 7;
    public static final short NOT_FOUND_ERR             = 8;
    public static final short NOT_SUPPORTED_ERR         = 9;
    public static final short INUSE_ATTRIBUTE_ERR       = 10;
    public static final short INVALID_STATE_ERR         = 11;
    public static final short SYNTAX_ERR                = 12;
    public static final short INVALID_MODIFICATION_ERR  = 13;
    public static final short NAMESPACE_ERR             = 14;
    public static final short INVALID_ACCESS_ERR        = 15;

}
```

org/w3c/dom/DOMImplementation.java:

```
package org.w3c.dom;

public interface DOMImplementation {
    public boolean hasFeature(String feature,
                              String version);

    public DocumentType createDocumentType(String qualifiedName,
                                           String publicId,
                                           String systemId)
                                           throws DOMException;

    public Document createDocument(String namespaceURI,
                                   String qualifiedName,
```

```
                                       DocumentType doctype)
                                       throws DOMException;

}
```

org/w3c/dom/DocumentFragment.java:

```
package org.w3c.dom;

public interface DocumentFragment extends Node {
}
```

org/w3c/dom/Document.java:

```
package org.w3c.dom;

public interface Document extends Node {
    public DocumentType getDoctype();

    public DOMImplementation getImplementation();

    public Element getDocumentElement();

    public Element createElement(String tagName)
                                 throws DOMException;

    public DocumentFragment createDocumentFragment();

    public Text createTextNode(String data);

    public Comment createComment(String data);

    public CDATASection createCDATASection(String data)
                                           throws DOMException;

    public ProcessingInstruction createProcessingInstruction(String target,
                                                             String data)
                                                             throws DOMException;

    public Attr createAttribute(String name)
                                throws DOMException;

    public EntityReference createEntityReference(String name)
                                                 throws DOMException;

    public NodeList getElementsByTagName(String tagname);

    public Node importNode(Node importedNode,
                           boolean deep)
                           throws DOMException;

    public Element createElementNS(String namespaceURI,
                                   String qualifiedName)
                                   throws DOMException;

    public Attr createAttributeNS(String namespaceURI,
```

```
                                  String qualifiedName)
                                  throws DOMException;

    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName);

    public Element getElementById(String elementId);

}
```

org/w3c/dom/Node.java:

```
package org.w3c.dom;

public interface Node {
    // NodeType
    public static final short ELEMENT_NODE              = 1;
    public static final short ATTRIBUTE_NODE            = 2;
    public static final short TEXT_NODE                 = 3;
    public static final short CDATA_SECTION_NODE        = 4;
    public static final short ENTITY_REFERENCE_NODE     = 5;
    public static final short ENTITY_NODE               = 6;
    public static final short PROCESSING_INSTRUCTION_NODE = 7;
    public static final short COMMENT_NODE              = 8;
    public static final short DOCUMENT_NODE             = 9;
    public static final short DOCUMENT_TYPE_NODE        = 10;
    public static final short DOCUMENT_FRAGMENT_NODE    = 11;
    public static final short NOTATION_NODE             = 12;

    public String getNodeName();

    public String getNodeValue()
                                  throws DOMException;
    public void setNodeValue(String nodeValue)
                                  throws DOMException;

    public short getNodeType();

    public Node getParentNode();

    public NodeList getChildNodes();

    public Node getFirstChild();

    public Node getLastChild();

    public Node getPreviousSibling();

    public Node getNextSibling();

    public NamedNodeMap getAttributes();

    public Document getOwnerDocument();

    public Node insertBefore(Node newChild,
                             Node refChild)
```

```
                             throws DOMException;

    public Node replaceChild(Node newChild,
                             Node oldChild)
                             throws DOMException;

    public Node removeChild(Node oldChild)
                            throws DOMException;

    public Node appendChild(Node newChild)
                            throws DOMException;

    public boolean hasChildNodes();

    public Node cloneNode(boolean deep);

    public void normalize();

    public boolean isSupported(String feature,
                               String version);

    public String getNamespaceURI();

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

    public String getLocalName();

    public boolean hasAttributes();

}
```

org/w3c/dom/NodeList.java:

```
package org.w3c.dom;

public interface NodeList {
    public Node item(int index);

    public int getLength();

}
```

org/w3c/dom/NamedNodeMap.java:

```
package org.w3c.dom;

public interface NamedNodeMap {
    public Node getNamedItem(String name);

    public Node setNamedItem(Node arg)
                             throws DOMException;

    public Node removeNamedItem(String name)
                                throws DOMException;
```

```
    public Node item(int index);

    public int getLength();

    public Node getNamedItemNS(String namespaceURI,
                               String localName);

    public Node setNamedItemNS(Node arg)
                               throws DOMException;

    public Node removeNamedItemNS(String namespaceURI,
                                  String localName)
                                  throws DOMException;

}
```

org/w3c/dom/CharacterData.java:

```
package org.w3c.dom;

public interface CharacterData extends Node {
    public String getData()
                                  throws DOMException;
    public void setData(String data)
                                  throws DOMException;

    public int getLength();

    public String substringData(int offset,
                                int count)
                                throws DOMException;

    public void appendData(String arg)
                           throws DOMException;

    public void insertData(int offset,
                           String arg)
                           throws DOMException;

    public void deleteData(int offset,
                           int count)
                           throws DOMException;

    public void replaceData(int offset,
                            int count,
                            String arg)
                            throws DOMException;

}
```

org/w3c/dom/Attr.java:

```
package org.w3c.dom;

public interface Attr extends Node {
    public String getName();

    public boolean getSpecified();

    public String getValue();
    public void setValue(String value)
                            throws DOMException;

    public Element getOwnerElement();

}
```

org/w3c/dom/Element.java:

```
package org.w3c.dom;

public interface Element extends Node {
    public String getTagName();

    public String getAttribute(String name);

    public void setAttribute(String name,
                             String value)
                             throws DOMException;

    public void removeAttribute(String name)
                                throws DOMException;

    public Attr getAttributeNode(String name);

    public Attr setAttributeNode(Attr newAttr)
                                 throws DOMException;

    public Attr removeAttributeNode(Attr oldAttr)
                                    throws DOMException;

    public NodeList getElementsByTagName(String name);

    public String getAttributeNS(String namespaceURI,
                                 String localName);

    public void setAttributeNS(String namespaceURI,
                               String qualifiedName,
                               String value)
                               throws DOMException;

    public void removeAttributeNS(String namespaceURI,
                                  String localName)
                                  throws DOMException;

    public Attr getAttributeNodeNS(String namespaceURI,
```

```
                                   String localName);

    public Attr setAttributeNodeNS(Attr newAttr)
                                   throws DOMException;

    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName);

    public boolean hasAttribute(String name);

    public boolean hasAttributeNS(String namespaceURI,
                                  String localName);

}
```

org/w3c/dom/Text.java:

```
package org.w3c.dom;

public interface Text extends CharacterData {
    public Text splitText(int offset)
                          throws DOMException;

}
```

org/w3c/dom/Comment.java:

```
package org.w3c.dom;

public interface Comment extends CharacterData {
}
```

org/w3c/dom/CDATASection.java:

```
package org.w3c.dom;

public interface CDATASection extends Text {
}
```

org/w3c/dom/DocumentType.java:

```
package org.w3c.dom;

public interface DocumentType extends Node {
    public String getName();

    public NamedNodeMap getEntities();

    public NamedNodeMap getNotations();

    public String getPublicId();

    public String getSystemId();
```

```
    public String getInternalSubset();

}
```

org/w3c/dom/Notation.java:

```
package org.w3c.dom;

public interface Notation extends Node {
    public String getPublicId();

    public String getSystemId();

}
```

org/w3c/dom/Entity.java:

```
package org.w3c.dom;

public interface Entity extends Node {
    public String getPublicId();

    public String getSystemId();

    public String getNotationName();

}
```

org/w3c/dom/EntityReference.java:

```
package org.w3c.dom;

public interface EntityReference extends Node {
}
```

org/w3c/dom/ProcessingInstruction.java:

```
package org.w3c.dom;

public interface ProcessingInstruction extends Node {
    public String getTarget();

    public String getData();
    public void setData(String data)
                          throws DOMException;

}
```

Appendix E: ECMA Script Language Binding

This appendix contains the complete ECMA Script [ECMAScript] binding for the Level 2 Document Object Model Core definitions.

Note: Exceptions handling is only supported by ECMAScript implementation compliant with the Standard ECMA-262 3rd. Edition ([ECMAScript]).

Class **DOMException**

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

DOMException.INDEX_SIZE_ERR
This constant is of type **short** and its value is **1**.

DOMException.DOMSTRING_SIZE_ERR
This constant is of type **short** and its value is **2**.

DOMException.HIERARCHY_REQUEST_ERR
This constant is of type **short** and its value is **3**.

DOMException.WRONG_DOCUMENT_ERR
This constant is of type **short** and its value is **4**.

DOMException.INVALID_CHARACTER_ERR
This constant is of type **short** and its value is **5**.

DOMException.NO_DATA_ALLOWED_ERR
This constant is of type **short** and its value is **6**.

DOMException.NO_MODIFICATION_ALLOWED_ERR
This constant is of type **short** and its value is **7**.

DOMException.NOT_FOUND_ERR
This constant is of type **short** and its value is **8**.

DOMException.NOT_SUPPORTED_ERR
This constant is of type **short** and its value is **9**.

DOMException.INUSE_ATTRIBUTE_ERR
This constant is of type **short** and its value is **10**.

DOMException.INVALID_STATE_ERR
This constant is of type **short** and its value is **11**.

DOMException.SYNTAX_ERR
This constant is of type **short** and its value is **12**.

DOMException.INVALID_MODIFICATION_ERR
This constant is of type **short** and its value is **13**.

DOMException.NAMESPACE_ERR
This constant is of type **short** and its value is **14**.

DOMException.INVALID_ACCESS_ERR
This constant is of type **short** and its value is **15**.

Exception **DOMException**

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

code
This property is of type **unsigned short**.

Object **DOMImplementation**

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

hasFeature(feature, version)

This method returns a **boolean**.
The **feature** parameter is of type **String**.
The **version** parameter is of type **String**.

createDocumentType(qualifiedName, publicId, systemId)

This method returns a **DocumentType**.
The **qualifiedName** parameter is of type **String**.
The **publicId** parameter is of type **String**.
The **systemId** parameter is of type **String**.
This method can raise a **DOMException**.

createDocument(namespaceURI, qualifiedName, doctype)

This method returns a **Document**.
The **namespaceURI** parameter is of type **String**.
The **qualifiedName** parameter is of type **String**.
The **doctype** parameter is of type **DocumentType**.
This method can raise a **DOMException**.

Object **DocumentFragment**

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

Object **Document**

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

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

doctype

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

implementation

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

documentElement

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

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

createElement(tagName)

This method returns a **Element**.
The **tagName** parameter is of type **String**.
This method can raise a **DOMException**.

createDocumentFragment()

This method returns a **DocumentFragment**.

createTextNode(data)

This method returns a **Text**.
The **data** parameter is of type **String**.

createComment(data)

This method returns a **Comment**.
The **data** parameter is of type **String**.

createCDATASection(data)

This method returns a **CDATASection**.

The **data** parameter is of type **String**.
This method can raise a **DOMException**.

createProcessingInstruction(target, data)
This method returns a **ProcessingInstruction**.
The **target** parameter is of type **String**.
The **data** parameter is of type **String**.
This method can raise a **DOMException**.

createAttribute(name)
This method returns a **Attr**.
The **name** parameter is of type **String**.
This method can raise a **DOMException**.

createEntityReference(name)
This method returns a **EntityReference**.
The **name** parameter is of type **String**.
This method can raise a **DOMException**.

getElementsByTagName(tagname)
This method returns a **NodeList**.
The **tagname** parameter is of type **String**.

importNode(importedNode, deep)
This method returns a **Node**.
The **importedNode** parameter is of type **Node**.
The **deep** parameter is of type **boolean**.
This method can raise a **DOMException**.

createElementNS(namespaceURI, qualifiedName)
This method returns a **Element**.
The **namespaceURI** parameter is of type **String**.
The **qualifiedName** parameter is of type **String**.
This method can raise a **DOMException**.

createAttributeNS(namespaceURI, qualifiedName)
This method returns a **Attr**.
The **namespaceURI** parameter is of type **String**.
The **qualifiedName** parameter is of type **String**.
This method can raise a **DOMException**.

getElementsByTagNameNS(namespaceURI, localName)
This method returns a **NodeList**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

getElementById(elementId)
This method returns a **Element**.
The **elementId** parameter is of type **String**.

Class **Node**

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

Node.ELEMENT_NODE
This constant is of type **short** and its value is **1**.

Node.ATTRIBUTE_NODE
This constant is of type **short** and its value is **2**.

Node.TEXT_NODE
This constant is of type **short** and its value is **3**.
Node.CDATA_SECTION_NODE
This constant is of type **short** and its value is **4**.
Node.ENTITY_REFERENCE_NODE
This constant is of type **short** and its value is **5**.
Node.ENTITY_NODE
This constant is of type **short** and its value is **6**.
Node.PROCESSING_INSTRUCTION_NODE
This constant is of type **short** and its value is **7**.
Node.COMMENT_NODE
This constant is of type **short** and its value is **8**.
Node.DOCUMENT_NODE
This constant is of type **short** and its value is **9**.
Node.DOCUMENT_TYPE_NODE
This constant is of type **short** and its value is **10**.
Node.DOCUMENT_FRAGMENT_NODE
This constant is of type **short** and its value is **11**.
Node.NOTATION_NODE
This constant is of type **short** and its value is **12**.

Object **Node**

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

nodeName
This read-only property is of type **String**.
nodeValue
This property is of type **String**, can raise a **DOMException** on setting and can raise a **DOMException** on retrieval.
nodeType
This read-only property is of type **short**.
parentNode
This read-only property is of type **Node**.
childNodes
This read-only property is of type **NodeList**.
firstChild
This read-only property is of type **Node**.
lastChild
This read-only property is of type **Node**.
previousSibling
This read-only property is of type **Node**.
nextSibling
This read-only property is of type **Node**.
attributes
This read-only property is of type **NamedNodeMap**.
ownerDocument
This read-only property is of type **Document**.

namespaceURI
This read-only property is of type **String**.
prefix
This property is of type **String** and can raise a **DOMException** on setting.
localName
This read-only property is of type **String**.
The **Node** object has the following methods:
insertBefore(newChild, refChild)
This method returns a **Node**.
The **newChild** parameter is of type **Node**.
The **refChild** parameter is of type **Node**.
This method can raise a **DOMException**.
replaceChild(newChild, oldChild)
This method returns a **Node**.
The **newChild** parameter is of type **Node**.
The **oldChild** parameter is of type **Node**.
This method can raise a **DOMException**.
removeChild(oldChild)
This method returns a **Node**.
The **oldChild** parameter is of type **Node**.
This method can raise a **DOMException**.
appendChild(newChild)
This method returns a **Node**.
The **newChild** parameter is of type **Node**.
This method can raise a **DOMException**.
hasChildNodes()
This method returns a **boolean**.
cloneNode(deep)
This method returns a **Node**.
The **deep** parameter is of type **boolean**.
normalize()
This method has no return value.
isSupported(feature, version)
This method returns a **boolean**.
The **feature** parameter is of type **String**.
The **version** parameter is of type **String**.
hasAttributes()
This method returns a **boolean**.
Object **NodeList**
The **NodeList** object has the following properties:
length
This read-only property is of type **int**.
The **NodeList** object has the following methods:
item(index)
This method returns a **Node**.
The **index** parameter is of type **int**.

Note: This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer **index** is equivalent to invoking the **item** method with that index.

Object **NamedNodeMap**

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

length

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

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

getNamedItem(name)

This method returns a **Node**.
The **name** parameter is of type **String**.

setNamedItem(arg)

This method returns a **Node**.
The **arg** parameter is of type **Node**.
This method can raise a **DOMException**.

removeNamedItem(name)

This method returns a **Node**.
The **name** parameter is of type **String**.
This method can raise a **DOMException**.

item(index)

This method returns a **Node**.
The **index** parameter is of type **int**.
Note: This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer **index** is equivalent to invoking the **item** method with that index.

getNamedItemNS(namespaceURI, localName)

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

setNamedItemNS(arg)

This method returns a **Node**.
The **arg** parameter is of type **Node**.
This method can raise a **DOMException**.

removeNamedItemNS(namespaceURI, localName)

This method returns a **Node**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.
This method can raise a **DOMException**.

Object **CharacterData**

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

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

data

This property is of type **String**, can raise a **DOMException** on setting and can raise a **DOMException** on retrieval.

length
This read-only property is of type **int**.

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

substringData(offset, count)
This method returns a **String**.
The **offset** parameter is of type **int**.
The **count** parameter is of type **int**.
This method can raise a **DOMException**.

appendData(arg)
This method has no return value.
The **arg** parameter is of type **String**.
This method can raise a **DOMException**.

insertData(offset, arg)
This method has no return value.
The **offset** parameter is of type **int**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException**.

deleteData(offset, count)
This method has no return value.
The **offset** parameter is of type **int**.
The **count** parameter is of type **int**.
This method can raise a **DOMException**.

replaceData(offset, count, arg)
This method has no return value.
The **offset** parameter is of type **int**.
The **count** parameter is of type **int**.
The **arg** parameter is of type **String**.
This method can raise a **DOMException**.

Object **Attr**

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

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

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

specified
This read-only property is of type **boolean**.

value
This property is of type **String** and can raise a **DOMException** on setting.

ownerElement
This read-only property is of type **Element**.

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

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

getAttribute(name)
This method returns a **String**.
The **name** parameter is of type **String**.

setAttribute(name, value)
This method has no return value.
The **name** parameter is of type **String**.
The **value** parameter is of type **String**.
This method can raise a **DOMException**.

removeAttribute(name)
This method has no return value.
The **name** parameter is of type **String**.
This method can raise a **DOMException**.

getAttributeNode(name)
This method returns a **Attr**.
The **name** parameter is of type **String**.

setAttributeNode(newAttr)
This method returns a **Attr**.
The **newAttr** parameter is of type **Attr**.
This method can raise a **DOMException**.

removeAttributeNode(oldAttr)
This method returns a **Attr**.
The **oldAttr** parameter is of type **Attr**.
This method can raise a **DOMException**.

getElementsByTagName(name)
This method returns a **NodeList**.
The **name** parameter is of type **String**.

getAttributeNS(namespaceURI, localName)
This method returns a **String**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

setAttributeNS(namespaceURI, qualifiedName, value)
This method has no return value.
The **namespaceURI** parameter is of type **String**.
The **qualifiedName** parameter is of type **String**.
The **value** parameter is of type **String**.
This method can raise a **DOMException**.

removeAttributeNS(namespaceURI, localName)
This method has no return value.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.
This method can raise a **DOMException**.

getAttributeNodeNS(namespaceURI, localName)
This method returns a **Attr**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

setAttributeNodeNS(newAttr)
This method returns a **Attr**.
The **newAttr** parameter is of type **Attr**.
This method can raise a **DOMException**.
getElementsByTagNameNS(namespaceURI, localName)
This method returns a **NodeList**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.
hasAttribute(name)
This method returns a **boolean**.
The **name** parameter is of type **String**.
hasAttributeNS(namespaceURI, localName)
This method returns a **boolean**.
The **namespaceURI** parameter is of type **String**.
The **localName** parameter is of type **String**.

Object **Text**
Text has the all the properties and methods of **CharacterData** as well as the properties and methods defined below.
The **Text** object has the following methods:
splitText(offset)
This method returns a **Text**.
The **offset** parameter is of type **int**.
This method can raise a **DOMException**.

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

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

Object **DocumentType**
DocumentType has the all the properties and methods of **Node** as well as the properties and methods defined below.
The **DocumentType** object has the following properties:
name
This read-only property is of type **String**.
entities
This read-only property is of type **NamedNodeMap**.
notations
This read-only property is of type **NamedNodeMap**.
publicId
This read-only property is of type **String**.
systemId
This read-only property is of type **String**.
internalSubset
This read-only property is of type **String**.

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:

publicId

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

systemId

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

Object **Entity**

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

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

publicId

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

systemId

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

notationName

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

Object **EntityReference**

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

Object **ProcessingInstruction**

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

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

target

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

data

This property is of type **String** and can raise a **DOMException** on setting.

Appendix F: Acknowledgements

Many people contributed to this specification, including members of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Lauren Wood (SoftQuad Software Inc., *chair*), Andrew Watson (Object Management Group), Andy Heninger (IBM), Arnaud Le Hors (W3C and IBM), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Singer (IBM), Don Park (invited), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégaret (W3C, *W3C team contact*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home and Netscape), Rich Rollman (Microsoft), Rick Gessner (Netscape), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tom Pixley (Netscape), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections.

F.1: Production Systems

This specification was written in XML. The HTML, OMG IDL, Java and ECMA Script bindings were all produced automatically.

Thanks to Joe English, author of cost, which was used as the basis for producing DOM Level 1. Thanks also to Gavin Nicol, who wrote the scripts which run on top of cost. Arnaud Le Hors and Philippe Le Hégaret maintained the scripts.

For DOM Level 2, we used Xerces as the basis DOM implementation and wish to thank the authors. Philippe Le Hégaret and Arnaud Le Hors wrote the Java programs which are the DOM application.

Thanks also to Jan Kärrman, author of html2ps, which we use in creating the PostScript version of the specification.

F.1: Production Systems

Glossary

Editors

Arnaud Le Hors, W3C and IBM
Lauren Wood, SoftQuad Software Inc.
Robert S. Sutor, IBM Research (for DOM Level 1)

Several of the following term definitions have been borrowed or modified from similar definitions in other W3C or standards documents. See the links within the definitions for more information.

16-bit unit
The base unit of a `DOMString` [p.17] . This indicates that indexing on a `DOMString` occurs in units of 16 bits. This must not be misunderstood to mean that a `DOMString` can store arbitrary 16-bit units. A `DOMString` is a character string encoded in UTF-16; this means that the restrictions of UTF-16 as well as the other relevant restrictions on character strings must be maintained. A single character, for example in the form of a numeric character reference, may correspond to one or two 16-bit units.
For more information, see [Unicode] and [ISO/IEC 10646].

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.

COM
COM is Microsoft's Component Object Model [COM], a technology for building applications from binary software components.

convenience
A *convenience method* is an operation on an object that could be accomplished by a program consisting of more basic operations on the object. Convenience *methods* are usually provided to make the API easier and simpler to use or to allow specific programs to create more optimized implementations for common operations. A similar definition holds for a *convenience property*.

data model
A *data model* is a collection of descriptions of data structures and their contained fields, together with the operations or functions that manipulate them.

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 [ECMAScript]. As stated in the standard, the originating technology for ECMAScript was JavaScript [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. See *Logical Structures* in XML [XML].

information item
: An information item is an abstract representation of some component of an XML document. See the [Infoset] for details.

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. [HTML4.0]

inheritance
: In object-oriented programming, the ability to create new classes (or interfaces) that contain all the methods and properties of another class (or interface), plus additional methods and properties. If class (or interface) D inherits from class (or interface) B, then D is said to be *derived* from B. B is said to be a *base* class (or interface) for D. Some programming languages allow for multiple inheritance, that is, inheritance from more than one class or interface.

interface
: An *interface* is a declaration of a set of *methods* with no information given about their implementation. In object systems that support interfaces and inheritance, interfaces can usually inherit from one another.

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.

local name
: A *local name* is the local part of a *qualified name*. This is called the local part in Namespaces in XML [Namespaces].

method
: A *method* is an operation or function that is associated with an object and is allowed to manipulate the object's data.

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.

namespace prefix
A *namespace prefix* is a string that associates an element or attribute name with a *namespace URI* in XML. See namespace prefix in Namespaces in XML [Namespaces].

namespace URI
A *namespace URI* is a URI that identifies an *XML namespace*. Strictly speaking, this actually is a *namespace URI reference*. This is called the namespace name in Namespaces in XML [Namespaces].

object model
An *object model* is a collection of descriptions of classes or interfaces, together with their member data, member functions, and class-static operations.

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

qualified name
A *qualified name* is the name of an element or attribute defined as the concatenation of a *local name* (as defined in this specification), optionally preceded by a *namespace prefix* and colon character. See *Qualified Names* in Namespaces in XML [Namespaces].

readonly node
A *readonly node* is a node that is immutable. This means its list of children, its content, and its attributes, when it is an element, cannot be changed in any way. However, a readonly node can possibly be moved, when it is not itself contained in a readonly 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 descendants of the root node. See *Well-Formed XML Documents* in XML [XML].

sibling
Two nodes are *siblings* if and only 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 3.0 standard [Unicode].

token
An information item such as an XML Name which has been *tokenized* [p.103] .

tokenized
The description given to various information items (for example, attribute values of various types, but not including the StringType CDATA) after having been processed by the XML processor. The process includes stripping leading and trailing white space, and replacing multiple space characters by one. See the definition of tokenized type.

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). See *Well-Formed XML Documents* in XML [XML].

XML
Extensible Markup Language (*XML*) is an extremely simple dialect of SGML. 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 [XML] has been designed for ease of implementation and for interoperability with both SGML and HTML.

XML name
See *XML name* in the XML specification [XML].

XML namespace
An *XML namespace* is a collection of names, identified by a URI reference [RFC2396], which are

used in XML documents as element types and attribute names. [Namespaces]

References

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

H.1: Normative references

Charmod
W3C (World Wide Web Consortium) Character Model for the World Wide Web, November 1999. Available at http://www.w3.org/TR/1999/WD-charmod-19991129

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

HTML4.0
W3C (World Wide Web Consortium) HTML 4.0 Specification, April 1998. Available at http://www.w3.org/TR/1998/REC-html40-19980424

ISO/IEC 10646
ISO (International Organization for Standardization). ISO/IEC 10646-1:2000 (E). Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization.

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

Namespaces
W3C (World Wide Web Consortium) Namespaces in XML, January 1999. Available at http://www.w3.org/TR/1999/REC-xml-names-19990114

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

RFC2396
IETF (Internet Engineering Task Force) RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, eds. T. Berners-Lee, R. Fielding, L. Masinter. August 1998. Available at http://www.ietf.org/rfc/rfc2396.txt

Unicode
The Unicode Consortium. The Unicode Standard, Version 3.0., February 2000. Available at http://www.unicode.org/unicode/standard/versions/Unicode3.0.html.

XML
W3C (World Wide Web Consortium) Extensible Markup Language (XML) 1.0, February 1998. Available at http://www.w3.org/TR/1998/REC-xml-19980210

H.2: Informative references

COM
Microsoft Corporation The Component Object Model. Available at http://www.microsoft.com/com

CORBA
OMG (Object Management Group) The Common Object Request Broker: Architecture and Specification, version 2.3.1, October 1999. Available from http://www.omg.org/

DOM Level 1
W3C (World Wide Web Consortium) DOM Level 1 Specification, October 1998. Available at http://www.w3.org/TR/REC-DOM-Level-1

DOM Level 2 HTML
W3C (World Wide Web Consortium) Document Object Model Level 2 HTML Specification, September 2000. Available at http://www.w3.org/TR/2000/PR-DOM-Level-2-HTML-20000927

Infoset
W3C (World Wide Web Consortium) XML Information Set, December 1999. Available at http://www.w3.org/TR/xml-infoset

JavaIDL
Sun Microsystems Inc. Java IDL. Available at http://java.sun.com/products/jdk/1.2/docs/guide/idl

JavaScript
Netscape Communications Corporation JavaScript Resources. Available at http://developer.netscape.com/tech/javascript/resources.html

JScript
Microsoft JScript Resources. Available at http://msdn.microsoft.com/scripting/default.htm

MIDL
Microsoft Corporation MIDL Language Reference. Available at http://msdn.microsoft.com/library/psdk/midl/mi-laref_1r1h.htm

XPointer
W3C (World Wide Web Consortium) XML Pointer Language (XPointer), June 2000. Available at http://www.w3.org/TR/xptr

Index