Document Object Model (DOM) Level 3 Core Specification

Version 1.0

W3C Recommendation 07 April 2004

This version:
http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407
Latest version:
http://www.w3.org/TR/DOM-Level-3-Core
Previous version:
http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205/

Editors:
Arnaud Le Hors, *IBM*
Philippe Le Hégaret, *W3C*
Lauren Wood, *SoftQuad, Inc. (WG Chair emerita, for DOM Level 1 and 2)*
Gavin Nicol, *Inso EPS (for DOM Level 1)*
Jonathan Robie, *Texcel Research and Software AG (for DOM Level 1 and 2)*
Mike Champion, *Arbortext and Software AG (for DOM Level 1 and 2)*
Steve Byrne, *JavaSoft (for DOM Level 1 until November 19, 1997)*

Please refer to the **errata** for this document, which may include some normative corrections.

This document is also available in these non-normative formats: XML file, plain text, PostScript file, PDF file, single HTML file, and ZIP file.

See also translations of this document.

Abstract

This specification defines the Document Object Model Core Level 3, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model Core Level 3 builds on the Document Object Model Core Level 2 [DOM Level 2 Core].

This version enhances DOM Level 2 Core by completing the mapping between DOM and the XML Information Set [XML Information Set], including the support for XML Base [XML Base], adding the ability to attach user information to DOM Nodes or to bootstrap a DOM implementation, providing mechanisms to resolve namespace prefixes or to manipulate "ID" attributes, giving to type information, etc.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document contains the Document Object Model Level 3 Core specification and is a W3C Recommendation. It has been produced as part of the W3C DOM Activity. The authors of this document are the DOM Working Group participants. For more information about DOM, readers can also refer to DOM FAQ and DOM Conformance Test Suites.

It is based on the feedback received during the Proposed Recommendation period. Changes since the Proposed Recommendation version and an implementation report are available. Please refer to the errata for this document, which may include some normative corrections.

Comments on this document should be sent to the public mailing list www-dom@w3.org (public archive).

This is a stable document and has been endorsed by the W3C Membership and the participants of the DOM working group. The English version of this specification is the only normative version. See also translations.

Patent disclosures relevant to this specification may be found on the Working Group’s patent disclosure page. This document has been produced under the 24 January 2002 CPP as amended by the W3C Patent Policy Transition Procedure. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of contents

Table of contents

Expanded Table of Contents

W3C Copyright Notices and Licenses

This document is published under the W3C® Document Copyright Notice and License [p.9] . The bindings within this document are published under the W3C® Software Copyright Notice and License [p.10] . The software license requires "Notice of any changes or modifications to the W3C files, including the date changes were made." Consequently, modified versions of the DOM bindings must document that they do not conform to the W3C standard; in the case of the IDL definitions, the pragma prefix can no longer be 'w3c.org'; in the case of the Java language binding, the package names can no longer be in the 'org.w3c' package.

W3C® Document Copyright Notice and License

Note: This section is a copy of the W3C® Document Notice and License and could be found at http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

Public documents on the W3C site are provided by the copyright holders under the following license. 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 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 (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231"
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/2002/copyright-software-20021231

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

This work (and included software, documentation such as READMEs, 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 copy, modify, and distribute 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:

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, notices, or terms and conditions. If none exist, the W3C® Short Software Notice [p.11] should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code.
3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

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

W3C® Short Software Notice

Note: This section is a copy of the W3C® Short Software Notice and could be found at http://www.w3.org/Consortium/Legal/2002/copyright-software-short-notice-20021231

Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. This work is distributed under the W3C® Software License [1] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[1] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

What is the Document Object Model?

Editors:
Philippe Le Hégaret, W3C
Lauren Wood, SoftQuad Software Inc. (for DOM Level 2)
Jonathan Robie, Texcel (for DOM Level 1)

Introduction

The Document Object Model (DOM) is an application programming interface (API [p.205]) for valid HTML [p.206] and well-formed XML [p.208] 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.206] 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.205] . 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 [*OMG IDL*], as defined in the CORBA 2.3.1 specification [*CORBA*]. In addition to the OMG IDL specification, we provide language bindings [p.207] for Java [*Java*] and ECMAScript [*ECMAScript*] (an industry-standard scripting language based on JavaScript [*JavaScript*] and JScript [*JScript*]). Because of language binding restrictions, a mapping has to be applied between the OMG IDL and the programming language in used. For example, while the DOM uses IDL attributes in the definition of interfaces, Java does not allow interfaces to contain attributes:

```
// example 1: removing the first child of an element using ECMAScript
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// example 2: removing the first child of an element using Java
mySecondTrElement.removeChild(mySecondTrElement.getFirstChild());
```

Note: OMG IDL is used only as a language-independent and implementation-neutral way to specify interfaces [p.206] . Various other IDLs could have been used ([*COM*], [*Java IDL*], [*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.205] for documents. It is based on an object structure that closely resembles the structure of the documents it models [p.207] . For instance, consider this table, taken from an XHTML 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>
```

A graphical representation of the DOM of the example table, with whitespaces in element content (often abusively called "ignorable whitespace") removed, is:

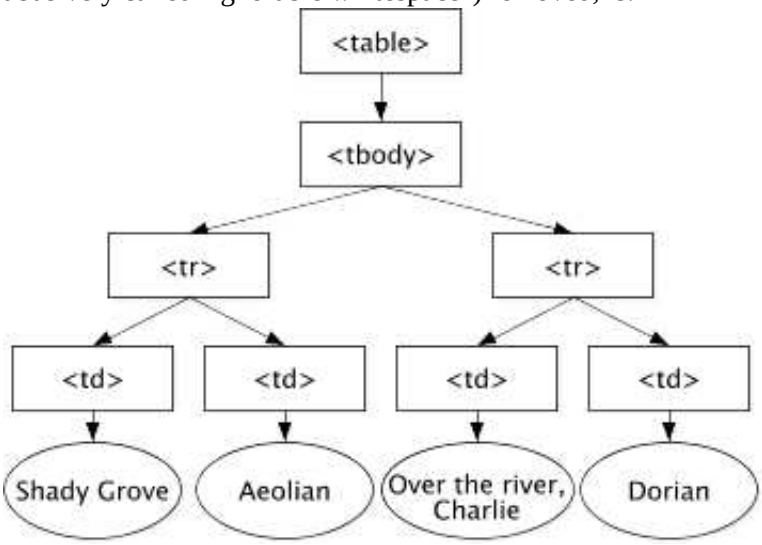

Figure: graphical representation of the DOM of the example table [SVG 1.0 version]

An example of DOM manipulation using ECMAScript would be:

```
// access the tbody element from the table element
var myTbodyElement = myTableElement.firstChild;

// access its second tr element
// The list of children starts at 0 (and not 1).
var mySecondTrElement = myTbodyElement.childNodes[1];

// remove its first td element
```

```
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// change the text content of the remaining td element
mySecondTrElement.firstChild.firstChild.data = "Peter";
```

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 document element node, and zero or more comments or processing instructions; the document 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 [*XML Information Set*].

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

The name "Document Object Model" was chosen because it is an "object model [p.207] " 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.205] , not by an object model. In an abstract data model [p.205] , 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.207] 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 XML Information Set [*XML Information Set*]. The DOM is simply an API [p.205] 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 Document Object Model Core [p.21] .

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.206] 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.

DOM Architecture

The DOM specifications provide a set of APIs that forms the DOM API. Each DOM specification defines one or more modules and each module is associated with one feature name. For example, the DOM Core specification (this specification) defines two modules:

- The Core module, which contains the fundamental interfaces that must be implemented by all DOM conformant implementations, is associated with the feature name "Core";
- The XML module, which contains the interfaces that must be implemented by all conformant XML 1.0 [*XML 1.0*] (and higher) DOM implementations, is associated with the feature name "XML".

The following representation contains all DOM modules, represented using their feature names, defined along the DOM specifications:

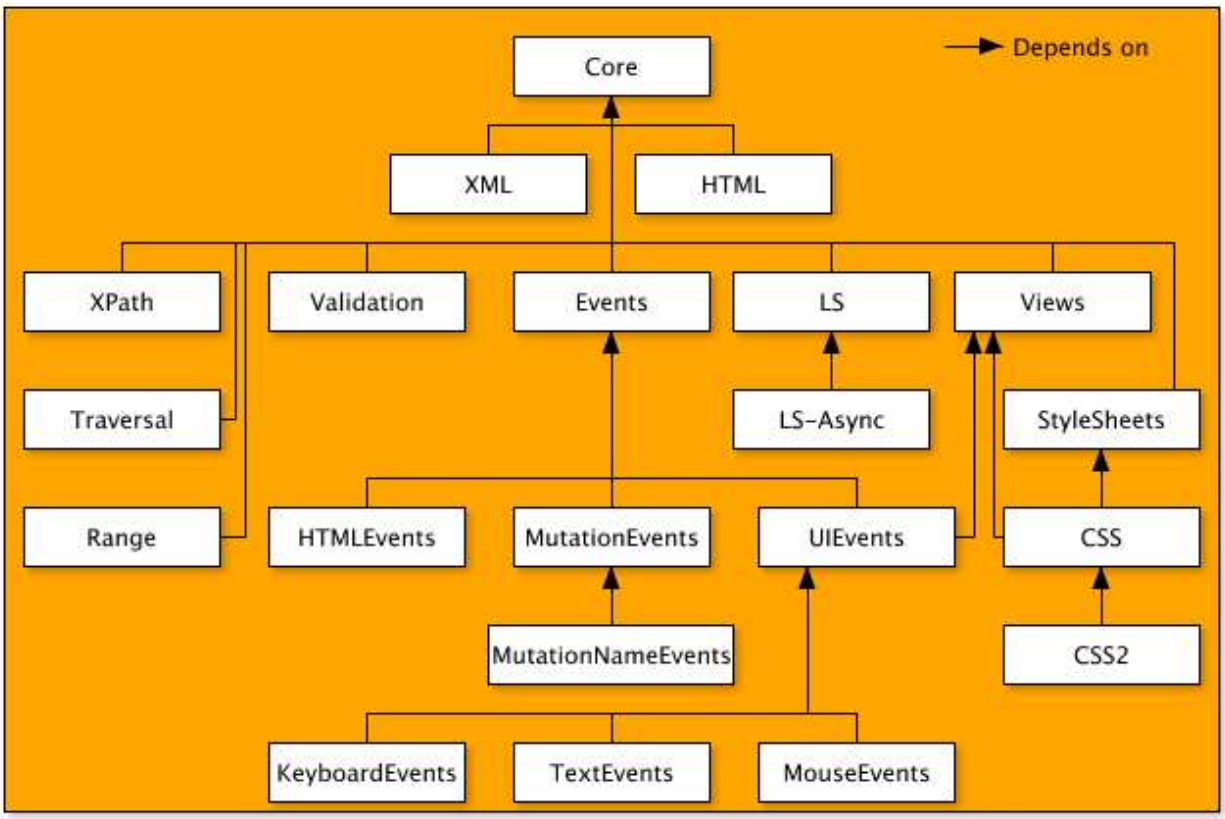

Figure: A view of the DOM Architecture [SVG 1.0 version]

A DOM implementation can then implement one (i.e. only the Core module) or more modules depending on the host application. A Web user agent is very likely to implement the "MouseEvents" module, while a server-side application will have no use of this module and will probably not implement it.

Conformance

This section explains the different levels of conformance to DOM Level 3. DOM Level 3 consists of 16 modules. It is possible to conform to DOM Level 3, or to a DOM Level 3 module.

An implementation is DOM Level 3 conformant if it supports the Core module defined in this document (see Fundamental Interfaces: Core Module [p.30]). An implementation conforms to a DOM Level 3 module if it supports all the interfaces for that module and the associated semantics.

Here is the complete list of DOM Level 3.0 modules and the features used by them. Feature names are case-insensitive.

Core module
: defines the feature *"Core"* [p.30] .

XML module
: Defines the feature *"XML"* [p.114] .

Events module
: defines the feature *"Events"* in [*DOM Level 3 Events*].

User interface Events module
: defines the feature *"UIEvents"* in [*DOM Level 3 Events*].

Mouse Events module
: defines the feature *"MouseEvents"* in [*DOM Level 3 Events*].

Text Events module
: defines the feature *"TextEvents"* in [*DOM Level 3 Events*].

Keyboard Events module
: defines the feature *"KeyboardEvents"* in [*DOM Level 3 Events*].

Mutation Events module
: defines the feature *"MutationEvents"* in [*DOM Level 3 Events*].

Mutation name Events module
: defines the feature *"MutationNameEvents"* in [*DOM Level 3 Events*].

HTML Events module
: defines the feature *"HTMLEvents"* in [*DOM Level 3 Events*].

Load and Save module
: defines the feature *"LS"* in [*DOM Level 3 Load and Save*].

Asynchronous load module
: defines the feature *"LS-Async"* in [*DOM Level 3 Load and Save*].

Validation module
: defines the feature *"Validation"* in [*DOM Level 3 Validation*].

XPath module
: defines the feature *"XPath"* in [*DOM Level 3 XPath*].

A DOM implementation must not return `true` to the `DOMImplementation.hasFeature(feature, version)` [p.40] method [p.207] of the `DOMImplementation` [p.37] interface for that feature unless the implementation conforms to that module. The `version` number for all features used in DOM Level 3.0 is `"3.0"`.

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 conformant.
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 2 interfaces were extended to provide both Level 2 and Level 3 functionality.

DOM implementations in languages other than Java or ECMAScript may choose bindings that are appropriate and natural for their language and run time environment. For example, some systems may need to create a Document3 class which inherits from a Document class and contains the new methods and attributes.

DOM Level 3 does not specify multithreading mechanisms.

1. Document Object Model Core

Editors:
Arnaud Le Hors, IBM
Philippe Le Hégaret, W3C
Gavin Nicol, Inso EPS (for DOM Level 1)
Lauren Wood, SoftQuad, Inc. (for DOM Level 1)
Mike Champion, Arbortext and Software AG (for DOM Level 1 from November 20, 1997)
Steve Byrne, JavaSoft (for DOM Level 1 until November 19, 1997)

This specification defines a set of objects and interfaces for accessing and manipulating document objects. The functionality specified (the *Core* functionality) is sufficient to allow software developers and Web script authors to access and manipulate parsed HTML [*HTML 4.01*] and XML [*XML 1.0*] content inside conforming products. The DOM Core API [p.205] also allows creation and population of a `Document` [p.41] object using only DOM API calls. A solution for loading a `Document` and saving it persistently is proposed in [*DOM Level 3 Load and Save*].

1.1 Overview of the DOM Core Interfaces

1.1.1 The DOM Structure Model

The DOM presents documents as a hierarchy of `Node` [p.56] objects that also implement other, more specialized interfaces. Some types of nodes may have child [p.205] 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.41] -- `Element` [p.85] (maximum of one), `ProcessingInstruction` [p.118] , `Comment` [p.99] , `DocumentType` [p.115] (maximum of one)
- `DocumentFragment` [p.40] -- `Element` [p.85] , `ProcessingInstruction` [p.118] , `Comment` [p.99] , `Text` [p.95] , `CDATASection` [p.114] , `EntityReference` [p.118]
- `DocumentType` [p.115] -- no children
- `EntityReference` [p.118] -- `Element` [p.85] , `ProcessingInstruction` [p.118] , `Comment` [p.99] , `Text` [p.95] , `CDATASection` [p.114] , `EntityReference`
- `Element` [p.85] -- `Element`, `Text` [p.95] , `Comment` [p.99] , `ProcessingInstruction` [p.118] , `CDATASection` [p.114] , `EntityReference` [p.118]
- `Attr` [p.81] -- `Text` [p.95] , `EntityReference` [p.118]
- `ProcessingInstruction` [p.118] -- no children
- `Comment` [p.99] -- no children
- `Text` [p.95] -- no children
- `CDATASection` [p.114] -- no children
- `Entity` [p.116] -- `Element` [p.85] , `ProcessingInstruction` [p.118] , `Comment` [p.99] , `Text` [p.95] , `CDATASection` [p.114] , `EntityReference` [p.118]
- `Notation` [p.116] -- no children

The DOM also specifies a `NodeList` [p.73] interface to handle ordered lists of `Nodes` [p.56] , such as the children of a `Node` [p.56] , or the elements [p.206] returned by the `Element.getElementsByTagNameNS(namespaceURI, localName)` [p.88] method, and also a `NamedNodeMap` [p.73] interface to handle unordered sets of nodes referenced by their name attribute, such as the attributes of an `Element` [p.85] . `NodeList` [p.73] and `NamedNodeMap` [p.73] 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.85] , then subsequently adds more children to that element [p.206] (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.56] in the tree are reflected in all references to that `Node` in `NodeList` and `NamedNodeMap` objects.

Finally, the interfaces `Text` [p.95] , `Comment` [p.99] , and `CDATASection` [p.114] all inherit from the `CharacterData` [p.78] 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.41] interface; this is because all DOM objects live in the context of a specific Document.

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 defined by the DOM API (for ECMAScript [p.206] 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 [*OMG IDL*] and ECMAScript [*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, DOM names tend to be long and 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, the DOM API uses 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.205] present two somewhat different sets of interfaces to an XML/HTML document: one presenting an "object oriented" approach with a hierarchy of inheritance [p.206] , and a "simplified" view that allows all manipulation to be done via the `Node` [p.56] interface without requiring casts (in Java and other C-like languages) or query interface calls in COM [p.205] 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.206] 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.205] .

In practice, this means that there is a certain amount of redundancy in the API [p.205] . The Working Group considers the "inheritance [p.206] " approach the primary view of the API, and the full set of functionality on `Node` [p.56] 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 `Node.nodeName` [p.62] attribute on the `Node` interface, there is still a `Element.tagName` [p.86] attribute on the `Element` [p.85] interface; these two attributes must contain the same value, but the it is worthwhile to support both, given the different constituencies the DOM API [p.205] must satisfy.

1.2 Basic Types

To ensure interoperability, this specification specifies the following basic types used in various DOM modules. Even though the DOM uses the basic types in the interfaces, bindings may use different types and normative bindings are only given for Java and ECMAScript in this specification.

1.2.1 The `DOMString` Type

The `DOMString` [p.24] type is used to store [*Unicode*] characters as a sequence of 16-bit units [p.205] using UTF-16 as defined in [*Unicode*] and Amendment 1 of [*ISO/IEC 10646*].

Characters are fully normalized as defined in appendix B of [*XML 1.1*] if:

- the parameter "normalize-characters [p.109] " was set to `true` while loading the document or the document was certified as defined in [*XML 1.1*];
- the parameter "normalize-characters [p.109] " was set to `true` while using the method `Document.normalizeDocument()` [p.54] , or while using the method `Node.normalize()` [p.71] ;

Note that, with the exceptions of `Document.normalizeDocument()` [p.54] and `Node.normalize()` [p.71] , manipulating characters using DOM methods does not guarantee to preserve a *fully-normalized* text.

Type Definition *DOMString*

A `DOMString` [p.24] is a sequence of 16-bit units [p.205] .
IDL Definition

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

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/IEC 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.24] (a high surrogate and a low surrogate). For issues related to string comparisons, refer to String Comparisons in the DOM [p.25] .

For Java and ECMAScript, `DOMString` [p.24] is bound to the `String` type because both languages also use UTF-16 as their encoding.

Note: As of August 2000, the OMG IDL specification ([*OMG IDL*]) included a `wstring` type. However, that definition did not meet the interoperability criteria of the DOM API [p.205] since it relied on negotiation to decide the width and encoding of a character.

1.2.2 The `DOMTimeStamp` Type

The `DOMTimeStamp` [p.24] type is used to store an absolute or relative time.

Type Definition *DOMTimeStamp*

A `DOMTimeStamp` [p.24] represents a number of milliseconds.
IDL Definition

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

For Java, `DOMTimeStamp` [p.24] is bound to the `long` type. For ECMAScript, `DOMTimeStamp` is bound to the `Date` type because the range of the `integer` type is too small.

1.2.3 The `DOMUserData` Type

The `DOMUserData` [p.25] type is used to store application data.

Type Definition *DOMUserData*

A `DOMUserData` [p.25] represents a reference to application data.
IDL Definition

```
typedef any DOMUserData;
```

For Java, `DOMUserData` [p.25] is bound to the `Object` type. For ECMAScript, `DOMUserData` is bound to `any type`.

1.2.4 The `DOMObject` Type

The `DOMObject` [p.25] type is used to represent an object.

Type Definition *DOMObject*

A `DOMObject` [p.25] represents an object reference.
IDL Definition

```
typedef Object DOMObject;
```

For Java and ECMAScript, `DOMObject` [p.25] is bound to the `Object` type.

1.3 General Considerations

1.3.1 String Comparisons in the DOM

The DOM has many interfaces that imply string matching. For XML, string comparisons are case-sensitive and performed with a binary comparison [p.208] of the 16-bit units [p.205] of the `DOMStrings` [p.24] . However, for case-insensitive markup languages, such as HTML 4.01 or earlier, these comparisons are case-insensitive where appropriate.

Note that HTML processors often perform specific case normalizations (canonicalization) of the markup before the DOM structures are built. This is typically using uppercase for element [p.206] names and lowercase for attribute names. For this reason, applications should also compare element and attribute names returned by the DOM implementation in a case-insensitive manner.

The character normalization, i.e. transforming into their fully normalized form as as defined in [*XML 1.1*], is assumed to happen at serialization time. The DOM Level 3 Load and Save module [*DOM Level 3 Load and Save*] provides a serialization mechanism (see the `DOMSerializer` interface, section 2.3.1) and uses the `DOMConfiguration` [p.106] parameters "normalize-characters [p.109] " and "check-character-normalization [p.107] " to assure that text is fully normalized [*XML 1.1*]. Other serialization mechanisms built on top of the DOM Level 3 Core also have to assure that text is *fully normalized*.

1.3.2 DOM URIs

The DOM specification relies on `DOMString` [p.24] values as resource identifiers, such that the following conditions are met:

1. An absolute identifier absolutely identifies a resource on the Web;
2. Simple string equality establishes equality of absolute resource identifiers, and no other equivalence of resource identifiers is considered significant to the DOM specification;
3. A relative identifier is easily detected and made absolute relative to an absolute identifier;
4. Retrieval of content of a resource may be accomplished where required.

The term "*absolute URI*" refers to a complete resource identifier and the term "*relative URI*" refers to an incomplete resource identifier.

Within the DOM specifications, these identifiers are called URIs, "Uniform Resource Identifiers", but this is meant abstractly. The DOM implementation does not necessarily process its URIs according to the URI specification [*IETF RFC 2396*]. Generally the particular form of these identifiers must be ignored.

When is not possible to completely ignore the type of a DOM URI, either because a relative identifier must be made absolute or because content must be retrieved, the DOM implementation must at least support identifier types appropriate to the content being processed. [*HTML 4.01*], [*XML 1.0*], and associated namespace specification [*XML Namespaces*] rely on [*IETF RFC 2396*] to determine permissible characters and resolving relative URIs. Other specifications such as namespaces in XML 1.1 [*XML Namespaces 1.1*] may rely on alternative resource identifier types that may, for example, include non-ASCII characters, necessitating support for alternative resource identifier types where required by applicable specifications.

1.3.3 XML Namespaces

DOM Level 2 and 3 support XML namespaces [*XML Namespaces*] by augmenting several interfaces of the DOM Level 1 Core to allow creating and manipulating elements [p.206] and attributes associated to a namespace. When [*XML 1.1*] is in use (see `Document.xmlVersion` [p.43]), DOM Level 3 also supports [*XML Namespaces 1.1*].

As far as the DOM is concerned, special attributes used for declaring XML namespaces are still exposed and can be manipulated just like any other attribute. However, nodes are permanently bound to namespace URIs [p.207] 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.207] 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.

In general, the DOM implementation (and higher) doesn't perform any URI normalization or canonicalization. The URIs given to the DOM are assumed to be valid (e.g., characters such as white spaces are properly escaped), and no lexical checking is performed. Absolute URI references are treated as strings and compared literally [p.208] . 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. Applications should use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace. In programming languages where empty strings can be differentiated from null, empty strings, when given as a namespace URI, are converted to `null`. This is true even though the DOM does no lexical checking of URIs.

Note: `Element.setAttributeNS(null, ...)` [p.91] puts the attribute in the *per-element-type partitions* as defined in *XML Namespace Partitions* in [*XML Namespaces*].

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.207] or qualified name [p.207] is "xmlns" as introduced in [*XML Namespaces 1.1*].

In a document with no namespaces, the child [p.205] list of an `EntityReference` [p.118] node is always the same as that of the corresponding `Entity` [p.116] . This is not true in a document where an entity contains unbound namespace prefixes [p.207] . In such a case, the descendants [p.205] of the corresponding `EntityReference` nodes may be bound to different namespace URIs [p.207] , 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 `Document.createEntityReference(name)` [p.49] is used to create entity references that correspond to such entities, since the descendants [p.205] of the returned `EntityReference` are unbound. While DOM Level 3 does have support for the resolution of namespace prefixes, use of such entities and entity references should be avoided or used with extreme care.

The "NS" methods, such as `Document.createElementNS(namespaceURI, qualifiedName)` [p.48] and `Document.createAttributeNS(namespaceURI, qualifiedName)` [p.46] , 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 `Document.createElement(tagName)` [p.48] and `Document.createAttribute(name)` [p.45] . 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 `Node.nodeName` [p.62] . On the contrary, the DOM Level 2 methods related to namespaces, identify attribute nodes by their

`Node.namespaceURI` [p.61] and `Node.localName` [p.61] . Because of this fundamental difference, mixing both sets of methods can lead to unpredictable results. In particular, using `Element.setAttributeNS(namespaceURI, qualifiedName, value)` [p.91] , an element [p.206] may have two attributes (or more) that have the same `Node.nodeName`, but different `Node.namespaceURI`s. Calling `Element.getAttribute(name)` [p.86] with that `nodeName` could then return any of those attributes. The result depends on the implementation. Similarly, using `Element.setAttributeNode(newAttr)` [p.92] , one can set two attributes (or more) that have different `Node.nodeName`s but the same `Node.prefix` [p.62] and `Node.namespaceURI`. In this case `Element.getAttributeNodeNS(namespaceURI, localName)` [p.87] 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, `Element.setAttribute(name, value)` [p.91] and `Element.setAttributeNS(namespaceURI, qualifiedName, value)` [p.91] affect the node that `Element.getAttribute(name)` [p.86] and `Element.getAttributeNS(namespaceURI, localName)` [p.87] , respectively, return.

1.3.4 Base URIs

The DOM Level 3 adds support for the **[base URI]** property defined in [*XML Information Set*] by providing a new attribute on the `Node` [p.56] interface that exposes this information. However, unlike the `Node.namespaceURI` [p.61] attribute, the `Node.baseURI` [p.61] attribute is not a static piece of information that every node carries. Instead, it is a value that is dynamically computed according to [*XML Base*]. This means its value depends on the location of the node in the tree and moving the node from one place to another in the tree may affect its value. Other changes, such as adding or changing an `xml:base` attribute on the node being queried or one of its ancestors may also affect its value.

One consequence of this it that when external entity references are expanded while building a `Document` [p.41] one may need to add, or change, an xml:base attribute to the `Element` [p.85] nodes originally contained in the entity being expanded so that the `Node.baseURI` [p.61] returns the correct value. In the case of `ProcessingInstruction` [p.118] nodes originally contained in the entity being expanded the information is lost. [*DOM Level 3 Load and Save*] handles elements as described here and generates a warning in the latter case.

1.3.5 Mixed DOM Implementations

As new XML vocabularies are developed, those defining the vocabularies are also beginning to define specialized APIs for manipulating XML instances of those vocabularies. This is usually done by extending the DOM to provide interfaces and methods that perform operations frequently needed by their users. For example, the MathML [*MathML 2.0*] and SVG [*SVG 1.1*] specifications have developed DOM extensions to allow users to manipulate instances of these vocabularies using semantics appropriate to images and mathematics, respectively, as well as the generic DOM XML semantics. Instances of SVG or MathML are often embedded in XML documents conforming to a different schema such as XHTML.

While the Namespaces in XML specification [*XML Namespaces*] provides a mechanism for integrating these documents at the syntax level, it has become clear that the DOM Level 2 Recommendation [*DOM Level 2 Core*] is not rich enough to cover all the issues that have been encountered in having these different DOM implementations be used together in a single application. DOM Level 3 deals with the requirements brought about by embedding fragments written according to a specific markup language (the embedded component) in a document where the rest of the markup is not written according to that specific markup language (the host document). It does not deal with fragments embedded by reference or linking.

A DOM implementation supporting DOM Level 3 Core should be able to collaborate with subcomponents implementing specific DOMs to assemble a compound document that can be traversed and manipulated via DOM interfaces as if it were a seamless whole.

The normal typecast operation on an object should support the interfaces expected by legacy code for a given document type. Typecasting techniques may not be adequate for selecting between multiple DOM specializations of an object which were combined at run time, because they may not all be part of the same object as defined by the binding's object model. Conflicts are most obvious with the `Document` [p.41] object, since it is shared as owner by the rest of the document. In a homogeneous document, elements rely on the Document for specialized services and construction of specialized nodes. In a heterogeneous document, elements from different modules expect different services and APIs from the same `Document` object, since there can only be one owner and root of the document hierarchy.

1.3.6 DOM Features

Each DOM module defines one or more features, as listed in the conformance section (Conformance [p.17]). Features are case-insensitive and are also defined for a specific set of versions. For example, this specification defines the features `"Core"` and `"XML"`, for the version `"3.0"`. Versions `"1.0"` and `"2.0"` can also be used for features defined in the corresponding DOM Levels. To avoid possible conflicts, as a convention, names referring to features defined outside the DOM specification should be made unique. Applications could then request for features to be supported by a DOM implementation using the methods `DOMImplementationSource.getDOMImplementation(features)` [p.36] or `DOMImplementationSource.getDOMImplementationList(features)` [p.37] , check the features supported by a DOM implementation using the method `DOMImplementation.hasFeature(feature, version)` [p.40] , or by a specific node using `Node.isSupported(feature, version)` [p.70] . Note that when using the methods that take a feature and a version as parameters, applications can use `null` or empty string for the version parameter if they don't wish to specify a particular version for the specified feature.

Up to the DOM Level 2 modules, all interfaces, that were an extension of existing ones, were accessible using binding-specific casting mechanisms if the feature associated to the extension was supported. For example, an instance of the `EventTarget` interface could be obtained from an instance of the `Node` [p.56] interface if the feature "Events" was supported by the node.

As discussed Mixed DOM Implementations [p.28] , DOM Level 3 Core should be able to collaborate with subcomponents implementing specific DOMs. For that effect, the methods `DOMImplementation.getFeature(feature, version)` [p.39] and `Node.getFeature(feature, version)` [p.66] were introduced. In the case of `DOMImplementation.hasFeature(feature, version)` [p.40] and

`Node.isSupported(feature, version)` [p.70] , if a plus sign "+" is prepended to any feature name, implementations are considered in which the specified feature may not be directly castable but would require discovery through `DOMImplementation.getFeature(feature, version)` [p.39] and `Node.getFeature(feature, version)` [p.66] . Without a plus, only features whose interfaces are directly castable are considered.

```
// example 1, without prepending the "+"
if (myNode.isSupported("Events", "3.0")) {
    EventTarget evt = (EventTarget) myNode;
    // ...
}
// example 2, with the "+"
if (myNode.isSupported("+Events", "3.0")) {
    // (the plus sign "+" is irrelevant for the getFeature method itself
    // and is ignored by this method anyway)
    EventTarget evt = (EventTarget) myNode.getFeature("Events", "3.0");
    // ...
}
```

1.3.7 Bootstrapping

Because previous versions of the DOM specification only defined a set of interfaces, applications had to rely on some implementation dependent code to start from. However, hard-coding the application to a specific implementation prevents the application from running on other implementations and from using the most-suitable implementation of the environment. At the same time, implementations may also need to load modules or perform other setup to efficiently adapt to different and sometimes mutually-exclusive feature sets.

To solve these problems this specification introduces a `DOMImplementationRegistry` object with a function that lets an application find implementations, based on the specific features it requires. How this object is found and what it exactly looks like is not defined here, because this cannot be done in a language-independent manner. Instead, each language binding defines its own way of doing this. See Java Language Binding [p.165] and ECMAScript Language Binding [p.185] for specifics.

In all cases, though, the `DOMImplementationRegistry` provides a `getDOMImplementation` method accepting a features string, which is passed to every known `DOMImplementationSource` [p.36] until a suitable `DOMImplementation` [p.37] is found and returned. The `DOMImplementationRegistry` also provides a `getDOMImplementationList` method accepting a features string, which is passed to every known `DOMImplementationSource`, and returns a list of suitable `DOMImplementations`. Those two methods are the same as the ones found on the `DOMImplementationSource` interface.

Any number of `DOMImplementationSource` [p.36] objects can be registered. A source may return one or more `DOMImplementation` [p.37] singletons or construct new `DOMImplementation` objects, depending upon whether the requested features require specialized state in the `DOMImplementation` object.

1.4 Fundamental Interfaces: Core Module

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.

A DOM application may use the `DOMImplementation.hasFeature(feature, version)` [p.40] method with parameter values "Core" and "3.0" (respectively) to determine whether or not this module is supported by the implementation. Any implementation that conforms to DOM Level 3 or a DOM Level 3 module must conform to the Core module. Please refer to additional information about *conformance* in this specification. The DOM Level 3 Core module is backward compatible with the DOM Level 2 Core [*DOM Level 2 Core*] module, i.e. a DOM Level 3 Core implementation who returns `true` for "Core" with the `version` number `"3.0"` must also return `true` for this `feature` when the `version` number is `"2.0"`, `""` or, `null`.

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.73] .

Implementations should raise other exceptions under other circumstances. For example, implementations should raise an implementation-dependent exception if a `null` argument is passed when `null` was not expected.

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;
// Introduced in DOM Level 3:
const unsigned short      VALIDATION_ERR                 = 16;
// Introduced in DOM Level 3:
const unsigned short      TYPE_MISMATCH_ERR              = 17;
```

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` [p.24] .

`HIERARCHY_REQUEST_ERR`
: If any `Node` [p.56] 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 an XML name.

`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` [p.56] in a context where it does not exist.

`NOT_SUPPORTED_ERR`
: If the implementation does not support the requested type of object or operation.

`NO_DATA_ALLOWED_ERR`
: If data is specified for a `Node` [p.56] 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.

`TYPE_MISMATCH_ERR`, introduced in **DOM Level 3**.
: If the type of an object is incompatible with the expected type of the parameter associated to the object.

`VALIDATION_ERR`, introduced in **DOM Level 3**.

If a call to a method such as `insertBefore` or `removeChild` would make the `Node` [p.56] invalid with respect to "partial validity" [p.207] , this exception would be raised and the operation would not be done. This code is used in [*DOM Level 3 Validation*]. Refer to this specification for further information.

`WRONG_DOCUMENT_ERR`

If a `Node` [p.56] is used in a different document than the one that created it (that doesn't support it).

Interface *DOMStringList* (introduced in **DOM Level 3**)

The `DOMStringList` interface provides the abstraction of an ordered collection of `DOMString` [p.24] values, without defining or constraining how this collection is implemented. The items in the `DOMStringList` are accessible via an integral index, starting from 0.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMStringList {
  DOMString          item(in unsigned long index);
  readonly attribute unsigned long   length;
  boolean            contains(in DOMString str);
};
```

Attributes

`length` of type `unsigned long`, readonly

The number of `DOMString` [p.24] s in the list. The range of valid child node indices is 0 to `length-1` inclusive.

Methods

`contains`

Test if a string is part of this `DOMStringList`.

Parameters

`str` of type `DOMString` [p.24]

The string to look for.

Return Value

`boolean`	`true` if the string has been found, `false` otherwise.

No Exceptions

`item`

Returns the `index`th item in the collection. If `index` is greater than or equal to the number of `DOMString` [p.24] s in the list, this returns `null`.

Parameters

`index` of type `unsigned long`

Index into the collection.

Return Value

`DOMString` [p.24]	The `DOMString` at the `index`th position in the `DOMStringList`, or `null` if that is not a valid index.

No Exceptions

Interface *NameList* (introduced in **DOM Level 3**)

The `NameList` interface provides the abstraction of an ordered collection of parallel pairs of name and namespace values (which could be null values), without defining or constraining how this collection is implemented. The items in the `NameList` are accessible via an integral index, starting from 0.

IDL Definition

```
// Introduced in DOM Level 3:
interface NameList {
  DOMString          getName(in unsigned long index);
  DOMString          getNamespaceURI(in unsigned long index);
  readonly attribute unsigned long   length;
  boolean            contains(in DOMString str);
  boolean            containsNS(in DOMString namespaceURI,
                                in DOMString name);
};
```

Attributes

`length` of type `unsigned long`, readonly

The number of pairs (name and namespaceURI) in the list. The range of valid child node indices is 0 to `length-1` inclusive.

Methods

`contains`

Test if a name is part of this `NameList`.

Parameters

`str` of type `DOMString` [p.24]

The name to look for.

Return Value

`boolean` `true` if the name has been found, `false` otherwise.

No Exceptions

`containsNS`

Test if the pair namespaceURI/name is part of this `NameList`.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI to look for.

`name` of type `DOMString`

The name to look for.

Return Value

`boolean` `true` if the pair namespaceURI/name has been found, `false` otherwise.

No Exceptions

`getName`

Returns the `index`th name item in the collection.

Parameters

`index` of type `unsigned long`

Index into the collection.

Return Value

`DOMString` [p.24]	The name at the `index`th position in the `NameList`, or `null` if there is no name for the specified index or if the index is out of range.

No Exceptions

`getNamespaceURI`

Returns the `index`th namespaceURI item in the collection.

Parameters

`index` of type `unsigned long`

Index into the collection.

Return Value

`DOMString` [p.24]	The namespace URI at the `index`th position in the `NameList`, or `null` if there is no name for the specified index or if the index is out of range.

No Exceptions

Interface *DOMImplementationList* (introduced in **DOM Level 3**)

The `DOMImplementationList` interface provides the abstraction of an ordered collection of DOM implementations, without defining or constraining how this collection is implemented. The items in the `DOMImplementationList` are accessible via an integral index, starting from 0.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMImplementationList {
  DOMImplementation  item(in unsigned long index);
  readonly attribute unsigned long   length;
};
```

Attributes

`length` of type `unsigned long`, readonly

The number of `DOMImplementation` [p.37] s 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 `DOMImplementation` [p.37] s in the list, this returns `null`.

Parameters

`index` of type `unsigned long`
Index into the collection.

Return Value

`DOMImplementation` [p.37]	The `DOMImplementation` at the `index`th position in the `DOMImplementationList`, or `null` if that is not a valid index.

No Exceptions

Interface *DOMImplementationSource* (introduced in **DOM Level 3**)

This interface permits a DOM implementer to supply one or more implementations, based upon requested features and versions, as specified in DOM Features [p.29] . Each implemented `DOMImplementationSource` object is listed in the binding-specific list of available sources so that its `DOMImplementation` [p.37] objects are made available.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMImplementationSource {
  DOMImplementation  getDOMImplementation(in DOMString features);
  DOMImplementationList getDOMImplementationList(in DOMString features);
};
```

Methods

`getDOMImplementation`
A method to request the first DOM implementation that supports the specified features.

Parameters

`features` of type `DOMString` [p.24]
A string that specifies which features and versions are required. This is a space separated list in which each feature is specified by its name optionally followed by a space and a version number.
This method returns the first item of the list returned by `getDOMImplementationList`.
As an example, the string `"XML 3.0 Traversal +Events 2.0"` will request a DOM implementation that supports the module "XML" for its 3.0 version, a module that support of the "Traversal" module for any version, and the module "Events" for its 2.0 version. The module "Events" must be accessible using the method `Node.getFeature()` [p.66] and `DOMImplementation.getFeature()` [p.39] .

Return Value

`DOMImplementation` [p.37]	The first DOM implementation that support the desired features, or `null` if this source has none.

No Exceptions

`getDOMImplementationList`

A method to request a list of DOM implementations that support the specified features and versions, as specified in DOM Features [p.29] .

Parameters

`features` of type `DOMString` [p.24]

A string that specifies which features and versions are required. This is a space separated list in which each feature is specified by its name optionally followed by a space and a version number. This is something like: "XML 3.0 Traversal +Events 2.0"

Return Value

`DOMImplementationList` [p.35]	A list of DOM implementations that support the desired features.

No Exceptions

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);
  // Introduced in DOM Level 3:
  DOMObject          getFeature(in DOMString feature,
                                in DOMString version);
};
```

Methods

`createDocument` introduced in **DOM Level 2**

Creates a DOM Document object of the specified type with its document element. Note that based on the `DocumentType` [p.115] given to create the document, the implementation may instantiate specialized `Document` [p.41] objects that support additional features than the "Core", such as "HTML" [*DOM Level 2 HTML*]. On the other hand, setting the `DocumentType` after the document was created makes this very unlikely to happen. Alternatively, specialized `Document` creation methods, such as `createHTMLDocument` [*DOM Level 2 HTML*], can be used to obtain specific types of `Document` objects.

Parameters

`namespaceURI` of type `DOMString` [p.24]
The namespace URI [p.207] of the document element to create or `null`.
`qualifiedName` of type `DOMString`
The qualified name [p.207] of the document element to be created or `null`.
`doctype` of type `DocumentType` [p.115]
The type of document to be created or `null`.
When `doctype` is not `null`, its `Node.ownerDocument` [p.62] attribute is set to the document being created.

Return Value

`Document` [p.41]	A new `Document` object with its document element. If the `NamespaceURI`, `qualifiedName`, and `doctype` are `null`, the returned `Document` is empty with no document element.

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified qualified name is not an XML name according to [*XML 1.0*].
	NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` is `null` and the `namespaceURI` is different from `null`, or if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from "http://www.w3.org/XML/1998/namespace" [*XML Namespaces*], or if the DOM implementation does not support the `"XML"` feature but a non-null namespace URI was provided, since namespaces were defined by XML.
	WRONG_DOCUMENT_ERR: Raised if `doctype` has already been used with a different document or was created from a different implementation.
	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature "XML" and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`createDocumentType` introduced in **DOM Level 2**
Creates an empty `DocumentType` [p.115] node. Entity declarations and notations are not made available. Entity reference expansions and default attribute additions do not occur..
Parameters
`qualifiedName` of type `DOMString` [p.24]
The qualified name [p.207] 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.115]	A new `DocumentType` node with `Node.ownerDocument` [p.62] set to `null`.

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified qualified name is not an XML name according to [*XML 1.0*].
	NAMESPACE_ERR: Raised if the `qualifiedName` is malformed.
	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature "XML" and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`getFeature` introduced in **DOM Level 3**
This method returns a specialized object which implements the specialized APIs of the specified feature and version, as specified in DOM Features [p.29] . The specialized object may also be obtained by using binding-specific casting methods but is not necessarily expected to, as discussed in Mixed DOM Implementations [p.28] . This method also allow the implementation to provide specialized objects which do not support the `DOMImplementation` interface.
Parameters
`feature` of type `DOMString` [p.24]
The name of the feature requested. Note that any plus sign "+" prepended to the name of the feature will be ignored since it is not significant in the context of this method.
`version` of type `DOMString`
This is the version number of the feature to test.
Return Value

`DOMObject` [p.25]	Returns an object which implements the specialized APIs of the specified feature and version, if any, or `null` if there is no object which implements interfaces associated with that feature. If the `DOMObject` returned by this method implements the `DOMImplementation` interface, it must delegate to the primary core `DOMImplementation` and not return results inconsistent with the primary core `DOMImplementation` such as `hasFeature`, `getFeature`, etc.

No Exceptions

`hasFeature`

Test if the DOM implementation implements a specific feature and version, as specified in DOM Features [p.29] .

Parameters

`feature` of type `DOMString` [p.24]

The name of the feature to test.

`version` of type `DOMString`

This is the version number of the feature to test.

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.41] 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.56] -- 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.208] (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.95] 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.41] (or indeed any other `Node` [p.56] 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.208] ; the `DocumentFragment` acts as the parent of these nodes so that the user can use the standard methods from the `Node` interface, such as `Node.insertBefore` [p.67] and `Node.appendChild` [p.64] .

IDL Definition

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

Interface *Document*

The `Document` interface represents the entire HTML or XML document. Conceptually, it is the root [p.207] 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.56] objects created have a `ownerDocument` attribute which associates them with the `Document` within whose context they were created.

IDL Definition

```
interface Document : Node {
  // Modified in DOM Level 3:
  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);
  // Introduced in DOM Level 3:
  readonly attribute DOMString       inputEncoding;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       xmlEncoding;
```

```
  // Introduced in DOM Level 3:
           attribute boolean         xmlStandalone;
                                        // raises(DOMException) on setting

  // Introduced in DOM Level 3:
           attribute DOMString       xmlVersion;
                                        // raises(DOMException) on setting

  // Introduced in DOM Level 3:
           attribute boolean         strictErrorChecking;
  // Introduced in DOM Level 3:
           attribute DOMString       documentURI;
  // Introduced in DOM Level 3:
  Node              adoptNode(in Node source)
                                        raises(DOMException);
  // Introduced in DOM Level 3:
  readonly attribute DOMConfiguration domConfig;
  // Introduced in DOM Level 3:
  void              normalizeDocument();
  // Introduced in DOM Level 3:
  Node              renameNode(in Node n,
                               in DOMString namespaceURI,
                               in DOMString qualifiedName)
                                        raises(DOMException);
};
```

Attributes

`doctype` of type `DocumentType` [p.115] , readonly, modified in **DOM Level 3**

The Document Type Declaration (see `DocumentType` [p.115]) associated with this document. For XML documents without a document type declaration this returns `null`. For HTML documents, a `DocumentType` object may be returned, independently of the presence or absence of document type declaration in the HTML document.

This provides direct access to the `DocumentType` [p.115] node, child node of this `Document`. This node can be set at document creation time and later changed through the use of child nodes manipulation methods, such as `Node.insertBefore` [p.67] , or `Node.replaceChild` [p.71] . Note, however, that while some implementations may instantiate different types of `Document` objects supporting additional features than the "Core", such as "HTML" [*DOM Level 2 HTML*], based on the `DocumentType` specified at creation time, changing it afterwards is very unlikely to result in a change of the features supported.

`documentElement` of type `Element` [p.85] , readonly

This is a convenience [p.205] attribute that allows direct access to the child node that is the document element [p.206] of the document.

`documentURI` of type `DOMString` [p.24] , introduced in **DOM Level 3**

The location of the document or `null` if undefined or if the `Document` was created using `DOMImplementation.createDocument` [p.37] . No lexical checking is performed when setting this attribute; this could result in a `null` value returned when using `Node.baseURI` [p.61] .

Beware that when the `Document` supports the feature "HTML" [*DOM Level 2 HTML*], the href attribute of the HTML BASE element takes precedence over this attribute when computing `Node.baseURI` [p.61] .

`domConfig` of type `DOMConfiguration` [p.106] , readonly, introduced in **DOM Level 3**
The configuration used when `Document.normalizeDocument()` [p.54] is invoked.

`implementation` of type `DOMImplementation` [p.37] , readonly
The `DOMImplementation` [p.37] object that handles this document. A DOM application may use objects from multiple implementations.

`inputEncoding` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
An attribute specifying the encoding used for this document at the time of the parsing. This is `null` when it is not known, such as when the `Document` was created in memory.

`strictErrorChecking` of type `boolean`, introduced in **DOM Level 3**
An attribute specifying whether error checking is enforced or not. When set to `false`, the implementation is free to not test every possible error case normally defined on DOM operations, and not raise any `DOMException` [p.31] on DOM operations or report errors while using `Document.normalizeDocument()` [p.54] . In case of error, the behavior is undefined. This attribute is `true` by default.

`xmlEncoding` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
An attribute specifying, as part of the *XML declaration*, the encoding of this document. This is `null` when unspecified or when it is not known, such as when the `Document` was created in memory.

`xmlStandalone` of type `boolean`, introduced in **DOM Level 3**
An attribute specifying, as part of the *XML declaration*, whether this document is standalone. This is `false` when unspecified.

Note: No verification is done on the value when setting this attribute. Applications should use `Document.normalizeDocument()` [p.54] with the "validate [p.110] " parameter to verify if the value matches the *validity constraint for standalone document declaration* as defined in [*XML 1.0*].

Exceptions on setting

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised if this document does not support the "XML" feature.

`xmlVersion` of type `DOMString` [p.24] , introduced in **DOM Level 3**
An attribute specifying, as part of the *XML declaration*, the version number of this document. If there is no declaration and if this document supports the "XML" feature, the value is `"1.0"`. If this document does not support the "XML" feature, the value is always `null`. Changing this attribute will affect methods that check for invalid characters in XML names. Application should invoke `Document.normalizeDocument()` [p.54] in order to check for invalid characters in the `Node` [p.56] s that are already part of this `Document`.
DOM applications may use the `DOMImplementation.hasFeature(feature, version)` [p.40] method with parameter values "XMLVersion" and "1.0" (respectively) to determine if an implementation supports [*XML 1.0*]. DOM applications may use the same method with parameter values "XMLVersion" and "1.1" (respectively) to determine if an implementation supports [*XML 1.1*]. In both cases, in order to support XML, an implementation must also support the "XML" feature defined in this specification.

`Document` objects supporting a version of the "XMLVersion" feature must not raise a `NOT_SUPPORTED_ERR` [p.32] exception for the same version number when using `Document.xmlVersion` [p.43] .

Exceptions on setting

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised if the version is set to a value that is not supported by this `Document` or if this document does not support the "XML" feature.

Methods

`adoptNode` introduced in **DOM Level 3**

Attempts to adopt a node from another document to this document. If supported, it changes the `ownerDocument` of the source node, its children, as well as the attached attribute nodes if there are any. If the source node has a parent it is first removed from the child list of its parent. This effectively allows moving a subtree from one document to another (unlike `importNode()` which create a copy of the source node instead of moving it). When it fails, applications should use `Document.importNode()` [p.52] instead. Note that if the adopted node is already part of this document (i.e. the source and target document are the same), this method still has the effect of removing the source node from the child list of its parent, if any. 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 adopted `Attr` [p.81] . The descendants of the source `Attr` are recursively adopted.

DOCUMENT_FRAGMENT_NODE
: The descendants of the source node are recursively adopted.

DOCUMENT_NODE
: `Document` nodes cannot be adopted.

DOCUMENT_TYPE_NODE
: `DocumentType` [p.115] nodes cannot be adopted.

ELEMENT_NODE
: *Specified* attribute nodes of the source element are adopted. Default attributes are discarded, though if the document being adopted into defines default attributes for this element name, those are assigned. The descendants of the source element are recursively adopted.

ENTITY_NODE
: `Entity` [p.116] nodes cannot be adopted.

ENTITY_REFERENCE_NODE
: Only the `EntityReference` [p.118] node itself is adopted, the descendants are discarded, 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.116] nodes cannot be adopted.

PROCESSING_INSTRUCTION_NODE, TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE
These nodes can all be adopted. No specifics.

Note: Since it does not create new nodes unlike the `Document.importNode()` [p.52] method, this method does not raise an `INVALID_CHARACTER_ERR` [p.32] exception, and applications should use the `Document.normalizeDocument()` [p.54] method to check if an imported name is not an XML name according to the XML version in use.

Parameters
`source` of type `Node` [p.56]
The node to move into this document.
Return Value

`Node` [p.56]	The adopted node, or `null` if this operation fails, such as when the source node comes from a different implementation.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised if the source node is of type `DOCUMENT`, `DOCUMENT_TYPE`. NO_MODIFICATION_ALLOWED_ERR: Raised when the source node is readonly.

`createAttribute`
Creates an `Attr` [p.81] of the given name. Note that the `Attr` instance can then be set on an `Element` [p.85] using the `setAttributeNode` method.
To create an attribute with a qualified name [p.207] and namespace URI [p.207] , use the `createAttributeNS` method.
Parameters
`name` of type `DOMString` [p.24]
The name of the attribute.
Return Value

`Attr` [p.81]	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.31]	INVALID_CHARACTER_ERR: Raised if the specified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute.

`createAttributeNS` introduced in **DOM Level 2**

Creates an attribute of the given qualified name [p.207] and namespace URI [p.207] . Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the attribute to create.

`qualifiedName` of type `DOMString`

The qualified name [p.207] of the attribute to instantiate.

Return Value

`Attr` [p.81] A new `Attr` object with the following attributes:

Attribute	Value
`Node.nodeName` [p.62]	qualifiedName
`Node.namespaceURI` [p.61]	`namespaceURI`
`Node.prefix` [p.62]	prefix, extracted from `qualifiedName`, or `null` if there is no prefix
`Node.localName` [p.61]	local name, extracted from `qualifiedName`
`Attr.name` [p.84]	`qualifiedName`
`Node.nodeValue` [p.62]	the empty string

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified `qualifiedName` is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. NAMESPACE_ERR: Raised if the `qualifiedName` is a malformed qualified name [p.207] , 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", if the `qualifiedName` or its prefix is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/", or if the `namespaceURI` is "http://www.w3.org/2000/xmlns/" and neither the `qualifiedName` nor its prefix is "xmlns". NOT_SUPPORTED_ERR: Always thrown if the current document does not support the `"XML"` feature, since namespaces were defined by XML.

`createCDATASection`

Creates a `CDATASection` [p.114] node whose value is the specified string.

Parameters

`data` of type `DOMString` [p.24]

The data for the `CDATASection` [p.114] contents.

Return Value

`CDATASection` [p.114]	The new `CDATASection` object.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createComment`

Creates a `Comment` [p.99] node given the specified string.

Parameters

`data` of type `DOMString` [p.24]

The data for the node.

Return Value

`Comment` [p.99]	The new `Comment` object.

No Exceptions

`createDocumentFragment`

Creates an empty `DocumentFragment` [p.40] object.

Return Value

`DocumentFragment` [p.40]	A new `DocumentFragment`.

No Parameters

No Exceptions

`createElement`

Creates an element of the type specified. Note that the instance returned implements the `Element` [p.85] interface, so attributes can be specified directly on the returned object. In addition, if there are known attributes with default values, `Attr` [p.81] nodes representing them are automatically created and attached to the element.

To create an element with a qualified name [p.207] and namespace URI [p.207] , use the `createElementNS` method.

Parameters

`tagName` of type `DOMString` [p.24]

The name of the element type to instantiate. For XML, this is case-sensitive, otherwise it depends on the case-sensitivity of the markup language in use. In that case, the name is mapped to the canonical form of that markup by the DOM implementation.

Return Value

`Element` [p.85]	A new `Element` object with the `nodeName` attribute set to `tagName`, and `localName`, `prefix`, and `namespaceURI` set to `null`.

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute.

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

Creates an element of the given qualified name [p.207] and namespace URI [p.207] . Per [*XML Namespaces*], applications must use the value `null` as the namespaceURI parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the element to create.

`qualifiedName` of type `DOMString`

The qualified name [p.207] of the element type to instantiate.

Return Value

`Element` [p.85]

A new `Element` object with the following attributes:

Attribute	Value
`Node.nodeName` [p.62]	`qualifiedName`
`Node.namespaceURI` [p.61]	`namespaceURI`
`Node.prefix` [p.62]	prefix, extracted from `qualifiedName`, or `null` if there is no prefix
`Node.localName` [p.61]	local name, extracted from `qualifiedName`
`Element.tagName` [p.86]	`qualifiedName`

Exceptions

`DOMException` [p.31]

INVALID_CHARACTER_ERR: Raised if the specified `qualifiedName` is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute.

NAMESPACE_ERR: Raised if the `qualifiedName` is a malformed qualified name [p.207] , 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" [*XML Namespaces*], or if the `qualifiedName` or its prefix is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/", or if the `namespaceURI` is "http://www.w3.org/2000/xmlns/" and neither the `qualifiedName` nor its prefix is "xmlns".

NOT_SUPPORTED_ERR: Always thrown if the current document does not support the `"XML"` feature, since namespaces were defined by XML.

`createEntityReference`

Creates an `EntityReference` [p.118] 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.116] node.

Note: If any descendant of the `Entity` [p.116] node has an unbound namespace prefix [p.207] , the corresponding descendant of the created `EntityReference` [p.118] node is also unbound; (its `namespaceURI` is `null`). The DOM Level 2 and 3 do not support any mechanism to resolve namespace prefixes in this case.

Parameters

`name` of type `DOMString` [p.24]
The name of the entity to reference.
Unlike `Document.createElementNS` [p.48] or `Document.createAttributeNS` [p.46] , no namespace well-formed checking is done on the entity name. Applications should invoke `Document.normalizeDocument()` [p.54] with the parameter "namespaces [p.109] " set to `true` in order to ensure that the entity name is namespace well-formed.

Return Value

`EntityReference` [p.118]	The new `EntityReference` object.

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createProcessingInstruction`

Creates a `ProcessingInstruction` [p.118] node given the specified name and data strings.

Parameters

`target` of type `DOMString` [p.24]
The target part of the processing instruction.
Unlike `Document.createElementNS` [p.48] or `Document.createAttributeNS` [p.46] , no namespace well-formed checking is done on the target name. Applications should invoke `Document.normalizeDocument()` [p.54] with the parameter "namespaces [p.109] " set to `true` in order to ensure that the target name is namespace well-formed.

`data` of type `DOMString`
The data for the node.

Return Value

`ProcessingInstruction` [p.118]	The new `ProcessingInstruction` object.

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified target is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createTextNode`

Creates a `Text` [p.95] node given the specified string.

Parameters

`data` of type `DOMString` [p.24]

The data for the node.

Return Value

`Text` [p.95]	The new `Text` object.

No Exceptions

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

Returns the `Element` [p.85] that has an ID attribute with the given value. If no such element exists, this returns `null`. If more than one element has an ID attribute with that value, what is returned is undefined.

The DOM implementation is expected to use the attribute `Attr.isId` [p.83] to determine if an attribute is of type ID.

Note: Attributes with the name "ID" or "id" are not of type ID unless so defined.

Parameters

`elementId` of type `DOMString` [p.24]

The unique `id` value for an element.

Return Value

`Element` [p.85]	The matching element or `null` if there is none.

No Exceptions

`getElementsByTagName`

Returns a `NodeList` [p.73] of all the `Elements` [p.85] in document order [p.206] with a given tag name and are contained in the document.

Parameters

`tagname` of type `DOMString` [p.24]

The name of the tag to match on. The special value "*" matches all tags. For XML, the `tagname` parameter is case-sensitive, otherwise it depends on the case-sensitivity of the markup language in use.

Return Value

`NodeList` [p.73]	A new `NodeList` object containing all the matched `Elements` [p.85] .

No Exceptions

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

Returns a `NodeList` [p.73] of all the `Elements` [p.85] with a given local name [p.207] and namespace URI [p.207] in document order [p.206] .

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the elements to match on. The special value `"*"` matches all namespaces.

`localName` of type `DOMString`

The local name [p.207] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.73]	A new `NodeList` object containing all the matched `Elements` [p.85] .

No Exceptions

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

Imports a node from another document to this document, without altering or removing the source node from the original document; this method creates a new copy of the source node. The returned node has no parent; (`parentNode` is `null`).

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, the source node is not altered. User data associated to the imported node is not carried over. However, if any `UserDataHandlers` [p.102] has been specified along with the associated data these handlers will be called with the appropriate parameters before this method returns.

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.81] . The descendants [p.205] 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.81] nodes; they always carry their children with them when imported.

DOCUMENT_FRAGMENT_NODE

If the `deep` option was set to `true`, the descendants [p.205] of the source `DocumentFragment` [p.40] are recursively imported and the resulting nodes

reassembled under the imported `DocumentFragment` to form the corresponding subtree. Otherwise, this simply generates an empty `DocumentFragment`.

DOCUMENT_NODE

`Document` nodes cannot be imported.

DOCUMENT_TYPE_NODE

`DocumentType` [p.115] nodes cannot be imported.

ELEMENT_NODE

Specified attribute nodes of the source element are imported, and the generated `Attr` [p.81] nodes are attached to the generated `Element` [p.85] . 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.205] of the source element are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_NODE

`Entity` [p.116] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.115] 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.205] of the the source `Entity` [p.116] are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_REFERENCE_NODE

Only the `EntityReference` [p.118] 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.116] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.115] 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 this type of nodes since they cannot have any children.

PROCESSING_INSTRUCTION_NODE

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

Note that the `deep` parameter has no effect on this type of nodes since they cannot have any children.

TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE

These three types of nodes inheriting from `CharacterData` [p.78] copy their `data` and `length` attributes from those of the source node.

Note that the `deep` parameter has no effect on these types of nodes since they cannot have any children.

Parameters

`importedNode` of type `Node` [p.56]

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 nodes that cannot have any children, and on `Attr` [p.81] , and `EntityReference` [p.118] nodes.

Return Value

`Node` [p.56]	The imported node that belongs to this `Document`.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised if the type of node being imported is not supported. INVALID_CHARACTER_ERR: Raised if one of the imported names is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. This may happen when importing an XML 1.1 [*XML 1.1*] element into an XML 1.0 document, for instance.

`normalizeDocument` introduced in **DOM Level 3**
This method acts as if the document was going through a save and load cycle, putting the document in a "normal" form. As a consequence, this method updates the replacement tree of `EntityReference` [p.118] nodes and normalizes `Text` [p.95] nodes, as defined in the method `Node.normalize()` [p.71] .
Otherwise, the actual result depends on the features being set on the `Document.domConfig` [p.43] object and governing what operations actually take place. Noticeably this method could also make the document namespace well-formed [p.207] according to the algorithm described in Namespace Normalization [p.125] , check the character normalization, remove the `CDATASection` [p.114] nodes, etc. See `DOMConfiguration` [p.106] for details.

```
// Keep in the document the information defined
// in the XML Information Set (Java example)
DOMConfiguration docConfig = myDocument.getDomConfig();
docConfig.setParameter("infoset", Boolean.TRUE);
myDocument.normalizeDocument();
```

Mutation events, when supported, are generated to reflect the changes occurring on the document.
If errors occur during the invocation of this method, such as an attempt to update a read-only node [p.207] or a `Node.nodeName` [p.62] contains an invalid character according to the XML version in use, errors or warnings (`DOMError.SEVERITY_ERROR` [p.104] or `DOMError.SEVERITY_WARNING` [p.104]) will be reported using the `DOMErrorHandler` [p.105] object associated with the "error-handler [p.108] " parameter. Note this method might also report fatal errors (`DOMError.SEVERITY_FATAL_ERROR` [p.104]) if an implementation cannot recover from an error.

No Parameters

No Return Value

No Exceptions

`renameNode` introduced in **DOM Level 3**

Rename an existing node of type `ELEMENT_NODE` or `ATTRIBUTE_NODE`.

When possible this simply changes the name of the given node, otherwise this creates a new node with the specified name and replaces the existing node with the new node as described below.

If simply changing the name of the given node is not possible, the following operations are performed: a new node is created, any registered event listener is registered on the new node, any user data attached to the old node is removed from that node, the old node is removed from its parent if it has one, the children are moved to the new node, if the renamed node is an `Element` [p.85] its attributes are moved to the new node, the new node is inserted at the position the old node used to have in its parent's child nodes list if it has one, the user data that was attached to the old node is attached to the new node.

When the node being renamed is an `Element` [p.85] only the specified attributes are moved, default attributes originated from the DTD are updated according to the new element name. In addition, the implementation may update default attributes from other schemas. Applications should use `Document.normalizeDocument()` [p.54] to guarantee these attributes are up-to-date.

When the node being renamed is an `Attr` [p.81] that is attached to an `Element` [p.85] , the node is first removed from the `Element` attributes map. Then, once renamed, either by modifying the existing node or creating a new one as described above, it is put back.

In addition,

- a user data event `NODE_RENAMED` is fired,
- when the implementation supports the feature "MutationNameEvents", each mutation operation involved in this method fires the appropriate event, and in the end the event {`http://www.w3.org/2001/xml-events`, `DOMElementNameChanged`} or {`http://www.w3.org/2001/xml-events`, `DOMAttributeNameChanged`} is fired.

Parameters

`n` of type `Node` [p.56]
: The node to rename.

`namespaceURI` of type `DOMString` [p.24]
: The new namespace URI [p.207] .

`qualifiedName` of type `DOMString`
: The new qualified name [p.207] .

Return Value

`Node` [p.56]	The renamed node. This is either the specified node or the new node that was created to replace the specified node.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: Raised when the type of the specified node is neither `ELEMENT_NODE` nor `ATTRIBUTE_NODE`, or if the implementation does not support the renaming of the document element [p.206] . INVALID_CHARACTER_ERR: Raised if the new qualified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. WRONG_DOCUMENT_ERR: Raised when the specified node was created from a different document than this document. NAMESPACE_ERR: Raised if the `qualifiedName` is a malformed qualified name [p.207] , 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" [*XML Namespaces*]. Also raised, when the node being renamed is an attribute, if the `qualifiedName`, or its prefix, is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/".

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.95] nodes may not have children, and adding children to such nodes results in a `DOMException` [p.31] 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.85] or `attributes` for a `Comment` [p.99]), 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;
// Modified in DOM Level 3:
Node               insertBefore(in Node newChild,
                                in Node refChild)
                                      raises(DOMException);
// Modified in DOM Level 3:
Node               replaceChild(in Node newChild,
                                in Node oldChild)
                                      raises(DOMException);
// Modified in DOM Level 3:
Node               removeChild(in Node oldChild)
                                      raises(DOMException);
// Modified in DOM Level 3:
Node               appendChild(in Node newChild)
                                      raises(DOMException);
boolean            hasChildNodes();
Node               cloneNode(in boolean deep);
// Modified in DOM Level 3:
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();
// Introduced in DOM Level 3:
readonly attribute DOMString       baseURI;

// DocumentPosition
const unsigned short      DOCUMENT_POSITION_DISCONNECTED = 0x01;
```

```
  const unsigned short      DOCUMENT_POSITION_PRECEDING    = 0x02;
  const unsigned short      DOCUMENT_POSITION_FOLLOWING    = 0x04;
  const unsigned short      DOCUMENT_POSITION_CONTAINS     = 0x08;
  const unsigned short      DOCUMENT_POSITION_CONTAINED_BY = 0x10;
  const unsigned short      DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;

  // Introduced in DOM Level 3:
  unsigned short     compareDocumentPosition(in Node other)
                                     raises(DOMException);
  // Introduced in DOM Level 3:
           attribute DOMString       textContent;
                                     // raises(DOMException) on setting
                                     // raises(DOMException) on retrieval

  // Introduced in DOM Level 3:
  boolean            isSameNode(in Node other);
  // Introduced in DOM Level 3:
  DOMString          lookupPrefix(in DOMString namespaceURI);
  // Introduced in DOM Level 3:
  boolean            isDefaultNamespace(in DOMString namespaceURI);
  // Introduced in DOM Level 3:
  DOMString          lookupNamespaceURI(in DOMString prefix);
  // Introduced in DOM Level 3:
  boolean            isEqualNode(in Node arg);
  // Introduced in DOM Level 3:
  DOMObject          getFeature(in DOMString feature,
                                in DOMString version);
  // Introduced in DOM Level 3:
  DOMUserData        setUserData(in DOMString key,
                                 in DOMUserData data,
                                 in UserDataHandler handler);
  // Introduced in DOM Level 3:
  DOMUserData        getUserData(in DOMString key);
};
```

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.81] .

`CDATA_SECTION_NODE`
: The node is a `CDATASection` [p.114] .

`COMMENT_NODE`
: The node is a `Comment` [p.99] .

`DOCUMENT_FRAGMENT_NODE`
: The node is a `DocumentFragment` [p.40] .

`DOCUMENT_NODE`
: The node is a `Document` [p.41] .

`DOCUMENT_TYPE_NODE`
: The node is a `DocumentType` [p.115] .

`ELEMENT_NODE`
: The node is an `Element` [p.85] .

`ENTITY_NODE`
: The node is an `Entity` [p.116] .

`ENTITY_REFERENCE_NODE`
: The node is an `EntityReference` [p.118] .

`NOTATION_NODE`
: The node is a `Notation` [p.116] .

`PROCESSING_INSTRUCTION_NODE`
: The node is a `ProcessingInstruction` [p.118] .

`TEXT_NODE`
: The node is a `Text` [p.95] node.

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

Interface	nodeName	nodeValue	attributes
`Attr` [p.81]	same as `Attr.name` [p.84]	same as `Attr.value` [p.84]	`null`
`CDATASection` [p.114]	`"#cdata-section"`	same as `CharacterData.data` [p.79] , the content of the CDATA Section	`null`
`Comment` [p.99]	`"#comment"`	same as `CharacterData.data` [p.79] , the content of the comment	`null`
`Document` [p.41]	`"#document"`	`null`	`null`
`DocumentFragment` [p.40]	`"#document-fragment"`	`null`	`null`
`DocumentType` [p.115]	same as `DocumentType.name` [p.116]	`null`	`null`
`Element` [p.85]	same as `Element.tagName` [p.86]	`null`	`NamedNodeMap` [p.73]
`Entity` [p.116]	entity name	`null`	`null`
`EntityReference` [p.118]	name of entity referenced	`null`	`null`
`Notation` [p.116]	notation name	`null`	`null`
`ProcessingInstruction` [p.118]	same as `ProcessingInstruction.target` [p.119]	same as `ProcessingInstruction.data` [p.119]	`null`
`Text` [p.95]	`"#text"`	same as `CharacterData.data` [p.79] , the content of the text node	`null`

Definition group *DocumentPosition*

A bitmask indicating the relative document position of a node with respect to another node.

If the two nodes being compared are the same node, then no flags are set on the return.

Otherwise, the order of two nodes is determined by looking for common containers -- containers which contain both. A node directly contains any child nodes. A node also directly contains any other nodes attached to it such as attributes contained in an element or entities and notations

contained in a document type. Nodes contained in contained nodes are also contained, but less-directly as the number of intervening containers increases.

If there is no common container node, then the order is based upon order between the root container of each node that is in no container. In this case, the result is disconnected and implementation-specific. This result is stable as long as these outer-most containing nodes remain in memory and are not inserted into some other containing node. This would be the case when the nodes belong to different documents or fragments, and cloning the document or inserting a fragment might change the order.

If one of the nodes being compared contains the other node, then the container precedes the contained node, and reversely the contained node follows the container. For example, when comparing an element against its own attribute or child, the element node precedes its attribute node and its child node, which both follow it.

If neither of the previous cases apply, then there exists a most-direct container common to both nodes being compared. In this case, the order is determined based upon the two determining nodes directly contained in this most-direct common container that either are or contain the corresponding nodes being compared.

If these two determining nodes are both child nodes, then the natural DOM order of these determining nodes within the containing node is returned as the order of the corresponding nodes. This would be the case, for example, when comparing two child elements of the same element.

If one of the two determining nodes is a child node and the other is not, then the corresponding node of the child node follows the corresponding node of the non-child node. This would be the case, for example, when comparing an attribute of an element with a child element of the same element.

If neither of the two determining node is a child node and one determining node has a greater value of `nodeType` than the other, then the corresponding node precedes the other. This would be the case, for example, when comparing an entity of a document type against a notation of the same document type.

If neither of the two determining node is a child node and `nodeType` is the same for both determining nodes, then an implementation-dependent order between the determining nodes is returned. This order is stable as long as no nodes of the same nodeType are inserted into or removed from the direct container. This would be the case, for example, when comparing two attributes of the same element, and inserting or removing additional attributes might change the order between existing attributes.

Defined Constants

`DOCUMENT_POSITION_CONTAINED_BY`
: The node is contained by the reference node. A node which is contained is always following, too.

`DOCUMENT_POSITION_CONTAINS`
: The node contains the reference node. A node which contains is always preceding, too.

`DOCUMENT_POSITION_DISCONNECTED`
: The two nodes are disconnected. Order between disconnected nodes is always implementation-specific.

`DOCUMENT_POSITION_FOLLOWING`
: The node follows the reference node.

`DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC`
: The determination of preceding versus following is implementation-specific.

`DOCUMENT_POSITION_PRECEDING`
: The second node precedes the reference node.

Attributes

`attributes` of type `NamedNodeMap` [p.73] , readonly
: A `NamedNodeMap` [p.73] containing the attributes of this node (if it is an `Element` [p.85]) or `null` otherwise.

`baseURI` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
: The absolute base URI of this node or `null` if the implementation wasn't able to obtain an absolute URI. This value is computed as described in Base URIs [p.28] . However, when the `Document` [p.41] supports the feature "HTML" [*DOM Level 2 HTML*], the base URI is computed using first the value of the href attribute of the HTML BASE element if any, and the value of the `documentURI` attribute from the `Document` interface otherwise.

`childNodes` of type `NodeList` [p.73] , readonly
: A `NodeList` [p.73] that contains all children of this node. If there are no children, this is a `NodeList` containing no nodes.

`firstChild` of type `Node` [p.56] , readonly
: The first child of this node. If there is no such node, this returns `null`.

`lastChild` of type `Node` [p.56] , readonly
: The last child of this node. If there is no such node, this returns `null`.

`localName` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 2**
: Returns the local part of the qualified name [p.207] 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 `Document.createElement()` [p.48] , this is always `null`.

`namespaceURI` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 2**
: The namespace URI [p.207] of this node, or `null` if it is unspecified (see XML Namespaces [p.26]).
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 `Document.createElement()` [p.48] , this is always `null`.

  Note: Per the *Namespaces in XML* Specification [*XML 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.56] , readonly
The node immediately following this node. If there is no such node, this returns `null`.
`nodeName` of type `DOMString` [p.24] , 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.24]
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, including if the node is read-only [p.207] .
Exceptions on setting

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly and if it is not defined to be `null`.

Exceptions on retrieval

`DOMException` [p.31]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.24] variable on the implementation platform.

`ownerDocument` of type `Document` [p.41] , readonly, modified in **DOM Level 2**
The `Document` [p.41] 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.115] which is not used with any `Document` yet, this is `null`.
`parentNode` of type `Node` [p.56] , readonly
The parent [p.207] of this node. All nodes, except `Attr` [p.81] , `Document` [p.41] , `DocumentFragment` [p.40] , `Entity` [p.116] , and `Notation` [p.116] 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.24] , introduced in **DOM Level 2**
The namespace prefix [p.207] of this node, or `null` if it is unspecified. When it is defined to be `null`, setting it has no effect, including if the node is read-only [p.207] .
Note that setting this attribute, when permitted, changes the `nodeName` attribute, which holds the qualified name [p.207] , as well as the `tagName` and `name` attributes of the `Element` [p.85] and `Attr` [p.81] interfaces, when applicable.
Setting the prefix to `null` makes it unspecified, setting it to an empty string is implementation dependent.
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.41] interface, this is always `null`.
Exceptions on setting

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified prefix contains an illegal character according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NAMESPACE_ERR: Raised if the specified `prefix` is malformed per the Namespaces in XML specification, 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" [*XML Namespaces*].

`previousSibling` of type `Node` [p.56] , readonly
: The node immediately preceding this node. If there is no such node, this returns `null`.

`textContent` of type `DOMString` [p.24] , introduced in **DOM Level 3**
: This attribute returns the text content of this node and its descendants. When it is defined to be `null`, setting it has no effect. On setting, any possible children this node may have are removed and, if it the new string is not empty or `null`, replaced by a single `Text` [p.95] node containing the string this attribute is set to.
On getting, no serialization is performed, the returned string does not contain any markup. No whitespace normalization is performed and the returned string does not contain the white spaces in element content (see the attribute `Text.isElementContentWhitespace` [p.96]). Similarly, on setting, no parsing is performed either, the input string is taken as pure textual content.
The string returned is made of the text content of this node depending on its type, as defined below:

Node type	Content
ELEMENT_NODE, ATTRIBUTE_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, DOCUMENT_FRAGMENT_NODE	concatenation of the `textContent` attribute value of every child node, excluding COMMENT_NODE and PROCESSING_INSTRUCTION_NODE nodes. This is the empty string if the node has no children.
TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE, PROCESSING_INSTRUCTION_NODE	`nodeValue`
DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE	*null*

Exceptions on setting

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.31]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.24] variable on the implementation platform.

Methods

`appendChild` modified in **DOM Level 3**

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.56]

The node to add.

If it is a `DocumentFragment` [p.40] object, the entire contents of the document fragment are moved into the child list of this node

Return Value

`Node` [p.56]	The node added.

Exceptions

`DOMException` [p.31]	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.205] or this node itself, or if this node is of type `Document` [p.41] and the DOM application attempts to append a second `DocumentType` [p.115] or `Element` [p.85] node. 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 previous parent of the node being inserted is readonly. NOT_SUPPORTED_ERR: if the `newChild` node is a child of the `Document` [p.41] node, this exception might be raised if the DOM implementation doesn't support the removal of the `DocumentType` [p.115] child or `Element` [p.85] child.

`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`) and no user data. User data associated to the imported node is not carried over. However, if any `UserDataHandlers` [p.102] has been specified along with the associated data these handlers will be called with the appropriate parameters before this method returns.
Cloning an `Element` [p.85] copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but this method does not copy any children it contains unless it is a deep clone. This includes text contained in an the `Element` since the text is contained in a child `Text` [p.95] node. Cloning an `Attr` [p.81] directly, as opposed to be cloned as part of an `Element` cloning operation, returns a specified attribute (`specified` is `true`). Cloning an `Attr` always clones its children, since they represent its value, no matter whether this is a deep clone or not. Cloning an `EntityReference` [p.118] automatically constructs its subtree if a corresponding `Entity` [p.116] is available, no matter whether this is a deep clone or not. 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.118] clone are readonly [p.207] . In addition, clones of unspecified `Attr` [p.81] nodes are specified. And, cloning `Document` [p.41] , `DocumentType` [p.115] , `Entity` [p.116] , and `Notation` [p.116] 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.85]).

Return Value

`Node` [p.56]	The duplicate node.

No Exceptions

`compareDocumentPosition` introduced in **DOM Level 3**

Compares the reference node, i.e. the node on which this method is being called, with a node, i.e. the one passed as a parameter, with regard to their position in the document and according to the document order [p.206] .

Parameters

`other` of type `Node` [p.56]

The node to compare against the reference node.

Return Value

`unsigned short`	Returns how the node is positioned relatively to the reference node.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: when the compared nodes are from different DOM implementations that do not coordinate to return consistent implementation-specific results.

`getFeature` introduced in **DOM Level 3**

This method returns a specialized object which implements the specialized APIs of the specified feature and version, as specified in DOM Features [p.29] . The specialized object may also be obtained by using binding-specific casting methods but is not necessarily expected to, as discussed in Mixed DOM Implementations [p.28] . This method also allow the implementation to provide specialized objects which do not support the `Node` interface.

Parameters

`feature` of type `DOMString` [p.24]

The name of the feature requested. Note that any plus sign "+" prepended to the name of the feature will be ignored since it is not significant in the context of this method.

`version` of type `DOMString`

This is the version number of the feature to test.

Return Value

`DOMObject` [p.25]	Returns an object which implements the specialized APIs of the specified feature and version, if any, or `null` if there is no object which implements interfaces associated with that feature. If the `DOMObject` returned by this method implements the `Node` interface, it must delegate to the primary core `Node` and not return results inconsistent with the primary core `Node` such as attributes, childNodes, etc.

No Exceptions

`getUserData` introduced in **DOM Level 3**

Retrieves the object associated to a key on a this node. The object must first have been set to this node by calling `setUserData` with the same key.

Parameters

`key` of type `DOMString` [p.24]

The key the object is associated to.

Return Value

`DOMUserData` [p.25]	Returns the `DOMUserData` associated to the given key on this node, or `null` if there was none.

No Exceptions

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

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

Return Value

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

No Parameters

No Exceptions

`hasChildNodes`

Returns whether this node has any children.

Return Value

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

No Parameters

No Exceptions

`insertBefore` modified in **DOM Level 3**

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.40] 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.

Note: Inserting a node before itself is implementation dependent.

Parameters

`newChild` of type `Node` [p.56]

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.56] The node being inserted.

Exceptions

`DOMException` [p.31]	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.205] or this node itself, or if this node is of type `Document` [p.41] and the DOM application attempts to insert a second `DocumentType` [p.115] or `Element` [p.85] node. 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. NOT_SUPPORTED_ERR: if this node is of type `Document` [p.41] , this exception might be raised if the DOM implementation doesn't support the insertion of a `DocumentType` [p.115] or `Element` [p.85] node.

`isDefaultNamespace` introduced in **DOM Level 3**

This method checks if the specified `namespaceURI` is the default namespace or not.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI to look for.

Return Value

`boolean`	Returns `true` if the specified `namespaceURI` is the default namespace, `false` otherwise.

No Exceptions

`isEqualNode` introduced in **DOM Level 3**

Tests whether two nodes are equal.

This method tests for equality of nodes, not sameness (i.e., whether the two nodes are references to the same object) which can be tested with `Node.isSameNode()` [p.69] . All nodes that are the same will also be equal, though the reverse may not be true.

Two nodes are equal if and only if the following conditions are satisfied:

- The two nodes are of the same type.
- The following string attributes are equal: `nodeName`, `localName`,

`namespaceURI`, `prefix`, `nodeValue`. This is: they are both `null`, or they have the same length and are character for character identical.

- The `attributes` `NamedNodeMaps` [p.73] are equal. This is: they are both `null`, or they have the same length and for each node that exists in one map there is a node that exists in the other map and is equal, although not necessarily at the same index.
- The `childNodes` `NodeLists` [p.73] are equal. This is: they are both `null`, or they have the same length and contain equal nodes at the same index. Note that normalization can affect equality; to avoid this, nodes should be normalized before being compared.

For two `DocumentType` [p.115] nodes to be equal, the following conditions must also be satisfied:

- The following string attributes are equal: `publicId`, `systemId`, `internalSubset`.
- The `entities` `NamedNodeMaps` [p.73] are equal.
- The `notations` `NamedNodeMaps` [p.73] are equal.

On the other hand, the following do not affect equality: the `ownerDocument`, `baseURI`, and `parentNode` attributes, the `specified` attribute for `Attr` [p.81] nodes, the `schemaTypeInfo` attribute for `Attr` and `Element` [p.85] nodes, the `Text.isElementContentWhitespace` [p.96] attribute for `Text` [p.95] nodes, as well as any user data or event listeners registered on the nodes.

Note: As a general rule, anything not mentioned in the description above is not significant in consideration of equality checking. Note that future versions of this specification may take into account more attributes and implementations conform to this specification are expected to be updated accordingly.

Parameters

`arg` of type `Node` [p.56]

The node to compare equality with.

Return Value

`boolean`	Returns `true` if the nodes are equal, `false` otherwise.

No Exceptions

`isSameNode` introduced in **DOM Level 3**

Returns whether this node is the same node as the given one.

This method provides a way to determine whether two `Node` references returned by the implementation reference the same object. When two `Node` references are references to the same object, even if through a proxy, the references may be used completely interchangeably, such that all attributes have the same values and calling the same DOM method on either reference always has exactly the same effect.

Parameters

`other` of type `Node` [p.56]

The node to test against.

Return Value

`boolean`	Returns `true` if the nodes are the same, `false` otherwise.

No Exceptions

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

Tests whether the DOM implementation implements a specific feature and that feature is supported by this node, as specified in DOM Features [p.29] .

Parameters

`feature` of type `DOMString` [p.24]

The name of the feature to test.

`version` of type `DOMString`

This is the version number of the feature to test.

Return Value

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

No Exceptions

`lookupNamespaceURI` introduced in **DOM Level 3**

Look up the namespace URI associated to the given prefix, starting from this node.
See Namespace URI Lookup [p.131] for details on the algorithm used by this method.

Parameters

`prefix` of type `DOMString` [p.24]

The prefix to look for. If this parameter is `null`, the method will return the default namespace URI if any.

Return Value

`DOMString` [p.24]	Returns the associated namespace URI or `null` if none is found.

No Exceptions

`lookupPrefix` introduced in **DOM Level 3**

Look up the prefix associated to the given namespace URI, starting from this node. The default namespace declarations are ignored by this method.
See Namespace Prefix Lookup [p.129] for details on the algorithm used by this method.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI to look for.

Return Value

`DOMString` [p.24]	Returns an associated namespace prefix if found or `null` if none is found. If more than one prefix are associated to the namespace prefix, the returned namespace prefix is implementation dependent.

No Exceptions

`normalize` modified in **DOM Level 3**

Puts all `Text` [p.95] 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. If the parameter "normalize-characters [p.109] " of the `DOMConfiguration` [p.106] object attached to the `Node.ownerDocument` [p.62] is `true`, this method will also fully normalize the characters of the `Text` nodes.

Note: In cases where the document contains `CDATASections` [p.114] , the normalize operation alone may not be sufficient, since XPointers do not differentiate between `Text` [p.95] nodes and `CDATASection` [p.114] nodes.

No Parameters

No Return Value

No Exceptions

`removeChild` modified in **DOM Level 3**

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

Parameters

`oldChild` of type `Node` [p.56]

The node being removed.

Return Value

`Node` [p.56]	The node removed.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.
	NOT_FOUND_ERR: Raised if `oldChild` is not a child of this node.
	NOT_SUPPORTED_ERR: if this node is of type `Document` [p.41] , this exception might be raised if the DOM implementation doesn’t support the removal of the `DocumentType` [p.115] child or the `Element` [p.85] child.

`replaceChild` modified in **DOM Level 3**

Replaces the child node `oldChild` with `newChild` in the list of children, and returns the `oldChild` node.

If `newChild` is a `DocumentFragment` [p.40] 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.

Note: Replacing a node with itself is implementation dependent.

Parameters

`newChild` of type `Node` [p.56]
The new node to put in the child list.

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

Return Value

`Node` [p.56]	The node replaced.

Exceptions

`DOMException` [p.31]	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.205] or this node itself, or if this node is of type `Document` [p.41] and the result of the replacement operation would add a second `DocumentType` [p.115] or `Element` [p.85] on the `Document` node. 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. NOT_SUPPORTED_ERR: if this node is of type `Document` [p.41] , this exception might be raised if the DOM implementation doesn't support the replacement of the `DocumentType` [p.115] child or `Element` [p.85] child.

`setUserData` introduced in **DOM Level 3**

Associate an object to a key on this node. The object can later be retrieved from this node by calling `getUserData` with the same key.

Parameters

`key` of type `DOMString` [p.24]
The key to associate the object to.

`data` of type `DOMUserData` [p.25]
The object to associate to the given key, or `null` to remove any existing association to that key.
`handler` of type `UserDataHandler` [p.102]
The handler to associate to that key, or `null`.

Return Value

`DOMUserData` [p.25]	Returns the `DOMUserData` previously associated to the given key on this node, or `null` if there was none.

No Exceptions

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.22] .

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.56]	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.73] ; `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.22] .

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)
                                     raises(DOMException);
  // 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.24]

The `nodeName` of a node to retrieve.

Return Value

`Node` [p.56]	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.

Per [*XML Namespaces*], applications must use the value null as the namespaceURI parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]
The namespace URI [p.207] of the node to retrieve.
`localName` of type `DOMString`
The local name [p.207] of the node to retrieve.
Return Value

`Node` [p.56]	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.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature "XML" and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`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.56]	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.24]
The `nodeName` of the node to remove.
Return Value

`Node` [p.56]	The node removed from this map if a node with such a name exists.

Exceptions

`DOMException` [p.31]	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.56] interface. If so, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.

Per [*XML Namespaces*], applications must use the value null as the namespaceURI parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the node to remove.

`localName` of type `DOMString`

The local name [p.207] of the node to remove.

Return Value

`Node` [p.56]	The node removed from this map if a node with such a local name and namespace URI exists.

Exceptions

`DOMException` [p.31]	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. NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature "XML" and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`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. Replacing a node by itself has no effect.

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.56]
A node to store in this map. The node will later be accessible using the value of its `nodeName` attribute.

Return Value

`Node` [p.56]	If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.31]	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.81] that is already an attribute of another `Element` [p.85] object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements. HIERARCHY_REQUEST_ERR: Raised if an attempt is made to add a node doesn't belong in this NamedNodeMap. Examples would include trying to insert something other than an Attr node into an Element's map of attributes, or a non-Entity node into the DocumentType's map of Entities.

`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. Replacing a node by itself has no effect.
Per [*XML Namespaces*], applications must use the value null as the namespaceURI parameter for methods if they wish to have no namespace.

Parameters

`arg` of type `Node` [p.56]
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.56]	If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.31]	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.81] that is already an attribute of another `Element` [p.85] object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements. HIERARCHY_REQUEST_ERR: Raised if an attempt is made to add a node doesn't belong in this NamedNodeMap. Examples would include trying to insert something other than an Attr node into an Element's map of attributes, or a non-Entity node into the DocumentType's map of Entities. NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature "XML" and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

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.95] and others do inherit the interface from it. All `offsets` in this interface start from `0`.

As explained in the `DOMString` [p.24] 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.205] 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.24]

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.24] . In such cases, the user may call `substringData` to retrieve the data in appropriately sized pieces.

Exceptions on setting

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.31]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.24] variable on the implementation platform.

`length` of type `unsigned long`, readonly

The number of 16-bit units [p.205] 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.24] specified.

Parameters

`arg` of type `DOMString` [p.24]

The `DOMString` to append.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`deleteData`

Remove a range of 16-bit units [p.205] 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.31]	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.205] offset.

Parameters

`offset` of type `unsigned long`

The character offset at which to insert.

`arg` of type `DOMString` [p.24]

The `DOMString` to insert.

Exceptions

`DOMException` [p.31]	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.205] 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.24]
The `DOMString` with which the range must be replaced.

Exceptions

`DOMException` [p.31]	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.24]	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.31]	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.24] .

Interface *Attr*

The `Attr` interface represents an attribute in an `Element` [p.85] object. Typically the allowable values for the attribute are defined in a schema associated with the document.

`Attr` objects inherit the `Node` [p.56] 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.40] . However, they can be associated with `Element` [p.85] 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 `Node.nodeValue` [p.62] attribute on the `Attr` instance can also be used to retrieve the string version of the attribute's value(s).

If the attribute was not explicitly given a value in the instance document but has a default value provided by the schema associated with the document, an attribute node will be created with `specified` set to `false`. Removing attribute nodes for which a default value is defined in the schema generates a new attribute node with the default value and `specified` set to `false`. If validation occurred while invoking `Document.normalizeDocument()` [p.54] , attribute nodes with `specified` equals to `false` are recomputed according to the default attribute values provided by the schema. If no default value is associate with this attribute in the schema, the attribute node is discarded.

In XML, where the value of an attribute can contain entity references, the child nodes of the `Attr` node may be either `Text` [p.95] or `EntityReference` [p.118] nodes (when these are in use; see the description of `EntityReference` for discussion).

The DOM Core represents all attribute values as simple strings, even if the DTD or schema associated with the document declares them of some specific type such as tokenized [p.208] .

The way attribute value normalization is performed by the DOM implementation depends on how much the implementation knows about the schema in use. Typically, the `value` and `nodeValue` attributes of an `Attr` node initially returns the normalized value given by the parser. It is also the case after `Document.normalizeDocument()` [p.54] is called (assuming the right options have been set). But this may not be the case after mutation, independently of whether the mutation is performed by setting the string value directly or by changing the `Attr` child nodes. In particular, this is true when *character references* are involved, given that they are not represented in the DOM and they impact attribute value normalization. On the other hand, if the implementation knows about the schema in use when the attribute value is changed, and it is of a different type than CDATA, it may normalize it again at that time. This is especially true of specialized DOM implementations, such as SVG DOM implementations, which store attribute values in an internal form different from a string.

The following table gives some examples of the relations between the attribute value in the original document (parsed attribute), the value as exposed in the DOM, and the serialization of the value:

Examples	Parsed attribute value	Initial `Attr.value` [p.84]	Serialized attribute value
Character reference	`"x²=5"`	`"x²=5"`	`"x²=5"`
Built-in character entity	`"y<6"`	`"y<6"`	`"y<6"`
Literal newline between	`"x=5
y=6"`	`"x=5` `y=6"`	`"x=5
y=6"`
Normalized newline between	`"x=5` `y=6"`	`"x=5 y=6"`	`"x=5 y=6"`
Entity `e` with literal newline	`<!ENTITY e '...
...'>` `[...]>` `"x=5&e;y=6"`	*Dependent on Implementation and Load Options*	*Dependent on Implementation and Load/Save Options*

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;
  // Introduced in DOM Level 3:
  readonly attribute TypeInfo        schemaTypeInfo;
  // Introduced in DOM Level 3:
  readonly attribute boolean         isId;
};
```

Attributes

`isId` of type `boolean`, readonly, introduced in **DOM Level 3**

Returns whether this attribute is known to be of type ID (i.e. to contain an identifier for its owner element) or not. When it is and its value is unique, the `ownerElement` of this attribute can be retrieved using the method `Document.getElementById` [p.51] . The implementation could use several ways to determine if an attribute node is known to contain an identifier:

- If validation occurred using an XML Schema [*XML Schema Part 1*] while loading the document or while invoking `Document.normalizeDocument()` [p.54] , the post-schema-validation infoset contributions (PSVI contributions) values are used to determine if this attribute is a *schema-determined ID attribute* using the schema-determined ID definition in [*XPointer*].
- If validation occurred using a DTD while loading the document or while invoking `Document.normalizeDocument()` [p.54] , the infoset **[type definition]** value is used to determine if this attribute is a *DTD-determined ID attribute* using the DTD-determined ID definition in [*XPointer*].

- from the use of the methods `Element.setIdAttribute()` [p.94] , `Element.setIdAttributeNS()` [p.94] , or `Element.setIdAttributeNode()` [p.95] , i.e. it is an *user-determined ID attribute*;

  Note: XPointer framework (see section 3.2 in [*XPointer*]) consider the DOM *user-determined ID attribute* as being part of the XPointer *externally-determined ID* definition.

- using mechanisms that are outside the scope of this specification, it is then an *externally-determined ID attribute*. This includes using schema languages different from XML schema and DTD.

If validation occurred while invoking `Document.normalizeDocument()` [p.54] , all *user-determined ID attributes* are reset and all attribute nodes ID information are then reevaluated in accordance to the schema used. As a consequence, if the `Attr.schemaTypeInfo` [p.84] attribute contains an ID type, `isId` will always return true.

`name` of type `DOMString` [p.24] , readonly
: Returns the name of this attribute. If `Node.localName` [p.61] is different from `null`, this attribute is a qualified name [p.207] .

`ownerElement` of type `Element` [p.85] , readonly, introduced in **DOM Level 2**
: The `Element` [p.85] node this attribute is attached to or `null` if this attribute is not in use.

`schemaTypeInfo` of type `TypeInfo` [p.99] , readonly, introduced in **DOM Level 3**
: The type information associated with this attribute. While the type information contained in this attribute is guarantee to be correct after loading the document or invoking `Document.normalizeDocument()` [p.54] , `schemaTypeInfo` may not be reliable if the node was moved.

`specified` of type `boolean`, readonly
: `True` if this attribute was explicitly given a value in the instance document, `false` otherwise. If the application changed the value of this attribute node (even if it ends up having the same value as the default value) then it is set to `true`. The implementation may handle attributes with default values from other schemas similarly but applications should use `Document.normalizeDocument()` [p.54] to guarantee this information is up-to-date.

`value` of type `DOMString` [p.24]
: 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.85] interface.
  On setting, this creates a `Text` [p.95] 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 `Element.setAttribute()` [p.91] .
  Some specialized implementations, such as some [*SVG 1.1*] implementations, may do normalization automatically, even after mutation; in such case, the value on retrieval may differ from the value on setting.
  Exceptions on setting

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Interface *Element*

The `Element` interface represents an element [p.206] in an HTML or XML document. Elements may have attributes associated with them; since the `Element` interface inherits from `Node` [p.56] , 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.81] 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.205] .

Note: In DOM Level 2, the method `normalize` is inherited from the `Node` [p.56] 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)
                                        raises(DOMException);
  // 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)
                                        raises(DOMException);
  // 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)
                                       raises(DOMException);
  // Introduced in DOM Level 2:
  boolean           hasAttribute(in DOMString name);
  // Introduced in DOM Level 2:
  boolean           hasAttributeNS(in DOMString namespaceURI,
                                   in DOMString localName)
                                       raises(DOMException);
  // Introduced in DOM Level 3:
  readonly attribute TypeInfo        schemaTypeInfo;
  // Introduced in DOM Level 3:
  void              setIdAttribute(in DOMString name,
                                   in boolean isId)
                                       raises(DOMException);
  // Introduced in DOM Level 3:
  void              setIdAttributeNS(in DOMString namespaceURI,
                                     in DOMString localName,
                                     in boolean isId)
                                       raises(DOMException);
  // Introduced in DOM Level 3:
  void              setIdAttributeNode(in Attr idAttr,
                                       in boolean isId)
                                       raises(DOMException);
};
```

Attributes

`schemaTypeInfo` of type `TypeInfo` [p.99] , readonly, introduced in **DOM Level 3**

The type information associated with this element.

`tagName` of type `DOMString` [p.24] , readonly

The name of the element. If `Node.localName` [p.61] is different from `null`, this attribute is a qualified name [p.207] . 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.24]

The name of the attribute to retrieve.

Return Value

`DOMString` [p.24]	The `Attr` [p.81] 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.

Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the attribute to retrieve.

`localName` of type `DOMString`

The local name [p.207] of the attribute to retrieve.

Return Value

`DOMString` [p.24]	The `Attr` [p.81] value as a string, or the empty string if that attribute does not have a specified or default value.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`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.24]

The name (`nodeName`) of the attribute to retrieve.

Return Value

`Attr` [p.81]	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.81] node by local name and namespace URI.

Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the attribute to retrieve.

`localName` of type `DOMString`

The local name [p.207] of the attribute to retrieve.

Return Value

`Attr` [p.81]	The `Attr` node with the specified attribute local name and namespace URI or `null` if there is no such attribute.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`getElementsByTagName`

Returns a `NodeList` [p.73] of all descendant [p.205] `Elements` with a given tag name, in document order [p.206] .

Parameters

`name` of type `DOMString` [p.24]

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

Return Value

`NodeList` [p.73]	A list of matching `Element` nodes.

No Exceptions

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

Returns a `NodeList` [p.73] of all the descendant [p.205] `Elements` with a given local name and namespace URI in document order [p.206] .

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the elements to match on. The special value "*" matches all namespaces.

`localName` of type `DOMString`

The local name [p.207] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.73]	A new `NodeList` object containing all the matched `Elements`.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`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.24]

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.

Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]

The namespace URI [p.207] of the attribute to look for.

`localName` of type `DOMString`

The local name [p.207] 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.

Exceptions

`DOMException` [p.31]	NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`removeAttribute`

Removes an attribute by name. If a default value for the removed attribute is defined in the DTD, a new attribute immediately appears with the default value as well as the corresponding namespace URI, local name, and prefix when applicable. The implementation may handle default values from other schemas similarly but applications should use `Document.normalizeDocument()` [p.54] to guarantee this information is up-to-date.

If no attribute with this name is found, this method has no effect.

To remove an attribute by local name and namespace URI, use the `removeAttributeNS` method.

Parameters

`name` of type `DOMString` [p.24]
The name of the attribute to remove.

Exceptions

`DOMException` [p.31]	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 a default value for the removed attribute is defined in the DTD, a new attribute immediately appears with the default value as well as the corresponding namespace URI, local name, and prefix when applicable. The implementation may handle default values from other schemas similarly but applications should use `Document.normalizeDocument()` [p.54] to guarantee this information is up-to-date.
If no attribute with this local name and namespace URI is found, this method has no effect.
Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.
Parameters
`namespaceURI` of type `DOMString` [p.24]
The namespace URI [p.207] of the attribute to remove.
`localName` of type `DOMString`
The local name [p.207] of the attribute to remove.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

No Return Value

`removeAttributeNode`
Removes the specified attribute node. If a default value for the removed `Attr` [p.81] node is defined in the DTD, a new node immediately appears with the default value as well as the corresponding namespace URI, local name, and prefix when applicable. The implementation may handle default values from other schemas similarly but applications should use `Document.normalizeDocument()` [p.54] to guarantee this information is up-to-date.
Parameters

`oldAttr` of type `Attr` [p.81]
The `Attr` node to remove from the attribute list.

Return Value

`Attr` [p.81] The `Attr` node that was removed.

Exceptions

`DOMException` [p.31]	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.81] node plus any `Text` [p.95] and `EntityReference` [p.118] 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.24]
The name of the attribute to create or alter.

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

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute.
	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.81] node plus any `Text` [p.95] and `EntityReference` [p.118] nodes, build the appropriate subtree, and use `setAttributeNodeNS` or `setAttributeNode` to assign it as the value of an attribute.
Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.

Parameters

`namespaceURI` of type `DOMString` [p.24]
The namespace URI [p.207] of the attribute to create or alter.

`qualifiedName` of type `DOMString`
The qualified name [p.207] of the attribute to create or alter.

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

Exceptions

`DOMException` [p.31]	INVALID_CHARACTER_ERR: Raised if the specified qualified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` [p.43] attribute. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed per the Namespaces in XML specification, 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", if the `qualifiedName` or its prefix is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/", or if the `namespaceURI` is "http://www.w3.org/2000/xmlns/" and neither the `qualifiedName` nor its prefix is "xmlns". NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

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. Replacing an attribute node by itself has no

effect.
To add a new attribute node with a qualified name and namespace URI, use the `setAttributeNodeNS` method.
Parameters
`newAttr` of type `Attr` [p.81]
The `Attr` node to add to the attribute list.
Return Value

`Attr` [p.81]	If the `newAttr` attribute replaces an existing attribute, the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.31]	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.81] 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. Replacing an attribute node by itself has no effect.
Per [*XML Namespaces*], applications must use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace.
Parameters
`newAttr` of type `Attr` [p.81]
The `Attr` node to add to the attribute list.
Return Value

`Attr` [p.81]	If the `newAttr` attribute replaces an existing attribute with the same local name [p.207] and namespace URI [p.207] , the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.31]	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.81] nodes to re-use them in other elements. NOT_SUPPORTED_ERR: May be raised if the implementation does not support the feature `"XML"` and the language exposed through the Document does not support XML Namespaces (such as [*HTML 4.01*]).

`setIdAttribute` introduced in **DOM Level 3**

If the parameter `isId` is `true`, this method declares the specified attribute to be a *user-determined ID attribute*. This affects the value of `Attr.isId` [p.83] and the behavior of `Document.getElementById` [p.51] , but does not change any schema that may be in use, in particular this does not affect the `Attr.schemaTypeInfo` [p.84] of the specified `Attr` [p.81] node. Use the value `false` for the parameter `isId` to undeclare an attribute for being a *user-determined ID attribute*.
To specify an attribute by local name and namespace URI, use the `setIdAttributeNS` method.

Parameters

`name` of type `DOMString` [p.24]
The name of the attribute.

`isId` of type `boolean`
Whether the attribute is a of type ID.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

No Return Value

`setIdAttributeNS` introduced in **DOM Level 3**

If the parameter `isId` is `true`, this method declares the specified attribute to be a *user-determined ID attribute*. This affects the value of `Attr.isId` [p.83] and the behavior of `Document.getElementById` [p.51] , but does not change any schema that may be in use, in particular this does not affect the `Attr.schemaTypeInfo` [p.84] of the specified `Attr` [p.81] node. Use the value `false` for the parameter `isId` to

undeclare an attribute for being a *user-determined ID attribute*.

Parameters

`namespaceURI` of type `DOMString` [p.24]
The namespace URI [p.207] of the attribute.

`localName` of type `DOMString`
The local name [p.207] of the attribute.

`isId` of type `boolean`
Whether the attribute is a of type ID.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.
	NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

No Return Value

`setIdAttributeNode` introduced in **DOM Level 3**

If the parameter `isId` is `true`, this method declares the specified attribute to be a *user-determined ID attribute*. This affects the value of `Attr.isId` [p.83] and the behavior of `Document.getElementById` [p.51] , but does not change any schema that may be in use, in particular this does not affect the `Attr.schemaTypeInfo` [p.84] of the specified `Attr` [p.81] node. Use the value `false` for the parameter `isId` to undeclare an attribute for being a *user-determined ID attribute*.

Parameters

`idAttr` of type `Attr` [p.81]
The attribute node.

`isId` of type `boolean`
Whether the attribute is a of type ID.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.
	NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

No Return Value

Interface *Text*

The `Text` interface inherits from `CharacterData` [p.78] and represents the textual content (termed *character data* in XML) of an `Element` [p.85] or `Attr` [p.81] . 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.206]

(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 `Node.normalize()` [p.71] method merges any such adjacent `Text` objects into a single node for each block of text.

No lexical check is done on the content of a `Text` node and, depending on its position in the document, some characters must be escaped during serialization using character references; e.g. the characters "<&" if the textual content is part of an element or of an attribute, the character sequence "]]>" when part of an element, the quotation mark character " or the apostrophe character ’ when part of an attribute.

IDL Definition

```
interface Text : CharacterData {
  Text               splitText(in unsigned long offset)
                                        raises(DOMException);
  // Introduced in DOM Level 3:
  readonly attribute boolean         isElementContentWhitespace;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       wholeText;
  // Introduced in DOM Level 3:
  Text               replaceWholeText(in DOMString content)
                                        raises(DOMException);
};
```

Attributes

`isElementContentWhitespace` of type `boolean`, readonly, introduced in **DOM Level 3**

Returns whether this text node contains *element content whitespace*, often abusively called "ignorable whitespace". The text node is determined to contain whitespace in element content during the load of the document or if validation occurs while using `Document.normalizeDocument()` [p.54] .

`wholeText` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**

Returns all text of `Text` nodes logically-adjacent text nodes [p.206] to this node, concatenated in document order.

For instance, in the example below `wholeText` on the `Text` node that contains "bar" returns "barfoo", while on the `Text` node that contains "foo" it returns "barfoo".

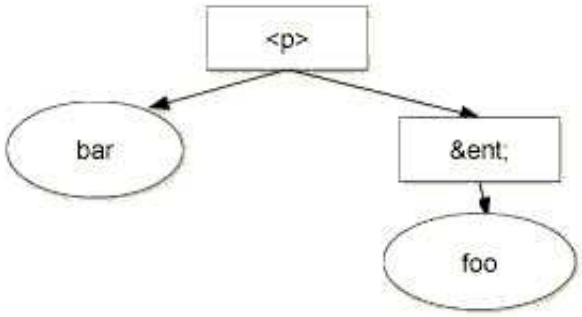

Figure: barTextNode.wholeText value is "barfoo" [SVG 1.0 version]

Methods

`replaceWholeText` introduced in **DOM Level 3**

Replaces the text of the current node and all logically-adjacent text nodes [p.206] with the specified text. All logically-adjacent text nodes [p.206] are removed including the current node unless it was the recipient of the replacement text.

This method returns the node which received the replacement text. The returned node is:

- `null`, when the replacement text is the empty string;
- the current node, except when the current node is read-only [p.207] ;
- a new `Text` node of the same type (`Text` or `CDATASection` [p.114]) as the current node inserted at the location of the replacement.

For instance, in the above example calling `replaceWholeText` on the `Text` node that contains "bar" with "yo" in argument results in the following:

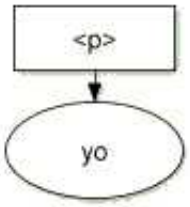

Figure: barTextNode.replaceWholeText("yo") modifies the textual content of barTextNode with "yo" [SVG 1.0 version]

Where the nodes to be removed are read-only descendants of an `EntityReference` [p.118] , the `EntityReference` must be removed instead of the read-only nodes. If any `EntityReference` to be removed has descendants that are not `EntityReference`, `Text`, or `CDATASection` [p.114] nodes, the `replaceWholeText` method must fail before performing any modification of the document, raising a `DOMException` [p.31] with the code `NO_MODIFICATION_ALLOWED_ERR` [p.32] .

For instance, in the example below calling `replaceWholeText` on the `Text` node that contains "bar" fails, because the `EntityReference` [p.118] node "ent" contains an `Element` [p.85] node which cannot be removed.

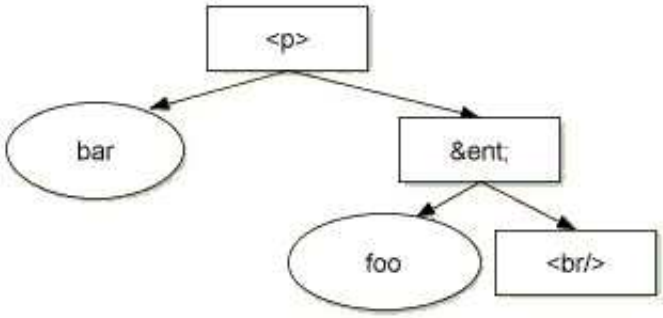

Figure: barTextNode.replaceWholeText("yo") raises a NO_MODIFICATION_ALLOWED_ERR DOMException [SVG 1.0 version]

Parameters

`content` of type `DOMString` [p.24]

The content of the replacing `Text` node.

Return Value

`Text` [p.95]	The `Text` node created with the specified content.

Exceptions

`DOMException` [p.31]	NO_MODIFICATION_ALLOWED_ERR: Raised if one of the `Text` nodes being replaced is readonly.

`splitText`

Breaks this node into two nodes at the specified `offset`, keeping both in the tree as siblings [p.208] . 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.208] 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.205] offset at which to split, starting from `0`.

Return Value

`Text` [p.95]	The new node, of the same type as this node.

Exceptions

`DOMException` [p.31]	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.78] 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.

No lexical check is done on the content of a comment and it is therefore possible to have the character sequence `"--"` (double-hyphen) in the content, which is illegal in a comment per section 2.5 of [*XML 1.0*]. The presence of this character sequence must generate a fatal error during serialization.

IDL Definition

```
interface Comment : CharacterData {
};
```

Interface *TypeInfo* (introduced in **DOM Level 3**)

The `TypeInfo` interface represents a type referenced from `Element` [p.85] or `Attr` [p.81] nodes, specified in the schemas [p.208] associated with the document. The type is a pair of a namespace URI [p.207] and name properties, and depends on the document's schema.

If the document's schema is an XML DTD [*XML 1.0*], the values are computed as follows:

- If this type is referenced from an `Attr` [p.81] node, `typeNamespace` is `"http://www.w3.org/TR/REC-xml"` and `typeName` represents the **[attribute type]** property in the [*XML Information Set*]. If there is no declaration for the attribute, `typeNamespace` and `typeName` are `null`.
- If this type is referenced from an `Element` [p.85] node, `typeNamespace` and `typeName` are `null`.

If the document's schema is an XML Schema [*XML Schema Part 1*], the values are computed as follows using the post-schema-validation infoset contributions (also called PSVI contributions):

- If the **[validity]** property exists AND is *"invalid"* or *"notKnown"*: the {target namespace} and {name} properties of the declared type if available, otherwise `null`.

  Note: At the time of writing, the XML Schema specification does not require exposing the declared type. Thus, DOM implementations might choose not to provide type information if validity is not valid.

- If the **[validity]** property exists and is *"valid"*:
  1. If **[member type definition]** exists:
     1. If {name} is not absent, then expose {name} and {target namespace} properties of the

[member type definition] property;
    2. Otherwise, expose the namespace and local name of the corresponding anonymous type name [p.205] .
2. If the **[type definition]** property exists:
    1. If {name} is not absent, then expose {name} and {target namespace} properties of the **[type definition]** property;
    2. Otherwise, expose the namespace and local name of the corresponding anonymous type name [p.205] .
3. If the **[member type definition anonymous]** exists:
    1. If it is false, then expose **[member type definition name]** and **[member type definition namespace]** properties;
    2. Otherwise, expose the namespace and local name of the corresponding anonymous type name [p.205] .
4. If the **[type definition anonymous]** exists:
    1. If it is false, then expose **[type definition name]** and **[type definition namespace]** properties;
    2. Otherwise, expose the namespace and local name of the corresponding anonymous type name [p.205] .

Note: Other schema languages are outside the scope of the W3C and therefore should define how to represent their type systems using `TypeInfo`.

IDL Definition

```
// Introduced in DOM Level 3:
interface TypeInfo {
  readonly attribute DOMString       typeName;
  readonly attribute DOMString       typeNamespace;

  // DerivationMethods
  const unsigned long       DERIVATION_RESTRICTION         = 0x00000001;
  const unsigned long       DERIVATION_EXTENSION           = 0x00000002;
  const unsigned long       DERIVATION_UNION               = 0x00000004;
  const unsigned long       DERIVATION_LIST                = 0x00000008;

  boolean            isDerivedFrom(in DOMString typeNamespaceArg,
                                   in DOMString typeNameArg,
                                   in unsigned long derivationMethod);
};
```

Definition group *DerivationMethods*

These are the available values for the `derivationMethod` parameter used by the method `TypeInfo.isDerivedFrom()` [p.102] . It is a set of possible types of derivation, and the values represent bit positions. If a bit in the `derivationMethod` parameter is set to `1`, the corresponding type of derivation will be taken into account when evaluating the derivation between the reference type definition and the other type definition. When using the `isDerivedFrom` method, combining all of them in the `derivationMethod` parameter is equivalent to invoking the method for each of them separately and combining the results with

the OR boolean function. This specification only defines the type of derivation for XML Schema.

In addition to the types of derivation listed below, please note that:

- any type derives from `xsd:anyType`.
- any simple type derives from `xsd:anySimpleType` by *restriction*.
- any complex type does not derive from `xsd:anySimpleType` by *restriction*.

Defined Constants

`DERIVATION_EXTENSION`

If the document's schema is an XML Schema [*XML Schema Part 1*], this constant represents the derivation by *extension*.

The reference type definition is derived by *extension* from the other type definition if the other type definition can be reached recursively following the {base type definition} property from the reference type definition, and at least one of the *derivation methods* involved is an *extension*.

`DERIVATION_LIST`

If the document's schema is an XML Schema [*XML Schema Part 1*], this constant represents the *list*.

The reference type definition is derived by *list* from the other type definition if there exists two type definitions T1 and T2 such as the reference type definition is derived from T1 by `DERIVATION_RESTRICTION` or `DERIVATION_EXTENSION`, T2 is derived from the other type definition by `DERIVATION_RESTRICTION`, T1 has {variety} *list*, and T2 is the {item type definition}. Note that T1 could be the same as the reference type definition, and T2 could be the same as the other type definition.

`DERIVATION_RESTRICTION`

If the document's schema is an XML Schema [*XML Schema Part 1*], this constant represents the derivation by *restriction* if complex types are involved, or a *restriction* if simple types are involved.

The reference type definition is derived by *restriction* from the other type definition if the other type definition is the same as the reference type definition, or if the other type definition can be reached recursively following the {base type definition} property from the reference type definition, and all the *derivation methods* involved are *restriction*.

`DERIVATION_UNION`

If the document's schema is an XML Schema [*XML Schema Part 1*], this constant represents the *union* if simple types are involved.

The reference type definition is derived by *union* from the other type definition if there exists two type definitions T1 and T2 such as the reference type definition is derived from T1 by `DERIVATION_RESTRICTION` or `DERIVATION_EXTENSION`, T2 is derived from the other type definition by `DERIVATION_RESTRICTION`, T1 has {variety} *union*, and one of the {member type definitions} is T2. Note that T1 could be the same as the reference type definition, and T2 could be the same as the other type definition.

Attributes

`typeName` of type `DOMString` [p.24] , readonly
: The name of a type declared for the associated element or attribute, or `null` if unknown.

`typeNamespace` of type `DOMString` [p.24] , readonly
: The namespace of the type declared for the associated element or attribute or `null` if the element does not have declaration or if no namespace information is available.

Methods

`isDerivedFrom`
: This method returns if there is a derivation between the reference type definition, i.e. the `TypeInfo` on which the method is being called, and the other type definition, i.e. the one passed as parameters.

Parameters

`typeNamespaceArg` of type `DOMString` [p.24]
: the namespace of the other type definition.

`typeNameArg` of type `DOMString`
: the name of the other type definition.

`derivationMethod` of type `unsigned long`
: the type of derivation and conditions applied between two types, as described in the list of constants provided in this interface.

Return Value

`boolean` If the document's schema is a DTD or no schema is associated with the document, this method will always return `false`.
If the document's schema is an XML Schema, the method will `true` if the reference type definition is derived from the other type definition according to the derivation parameter. If the value of the parameter is `0` (no bit is set to `1` for the `derivationMethod` parameter), the method will return `true` if the other type definition can be reached by recursing any combination of {base type definition}, {item type definition}, or {member type definitions} from the reference type definition.

No Exceptions

Interface *UserDataHandler* (introduced in **DOM Level 3**)

When associating an object to a key on a node using `Node.setUserData()` [p.72] the application can provide a handler that gets called when the node the object is associated to is being cloned, imported, or renamed. This can be used by the application to implement various behaviors regarding the data it associates to the DOM nodes. This interface defines that handler.

IDL Definition

```
// Introduced in DOM Level 3:
interface UserDataHandler {

  // OperationType
  const unsigned short      NODE_CLONED                    = 1;
  const unsigned short      NODE_IMPORTED                  = 2;
  const unsigned short      NODE_DELETED                   = 3;
  const unsigned short      NODE_RENAMED                   = 4;
  const unsigned short      NODE_ADOPTED                   = 5;
```

```
  void                handle(in unsigned short operation,
                             in DOMString key,
                             in DOMUserData data,
                             in Node src,
                             in Node dst);
};
```

Definition group *OperationType*

An integer indicating the type of operation being performed on a node.

Defined Constants

`NODE_ADOPTED`
: The node is adopted, using `Document.adoptNode()` [p.44] .

`NODE_CLONED`
: The node is cloned, using `Node.cloneNode()` [p.65] .

`NODE_DELETED`
: The node is deleted.

  Note: This may not be supported or may not be reliable in certain environments, such as Java, where the implementation has no real control over when objects are actually deleted.

`NODE_IMPORTED`
: The node is imported, using `Document.importNode()` [p.52] .

`NODE_RENAMED`
: The node is renamed, using `Document.renameNode()` [p.55] .

Methods

`handle`

This method is called whenever the node for which this handler is registered is imported or cloned.

DOM applications must not raise exceptions in a `UserDataHandler`. The effect of throwing exceptions from the handler is DOM implementation dependent.

Parameters

`operation` of type `unsigned short`
: Specifies the type of operation that is being performed on the node.

`key` of type `DOMString` [p.24]
: Specifies the key for which this handler is being called.

`data` of type `DOMUserData` [p.25]
: Specifies the data for which this handler is being called.

`src` of type `Node` [p.56]
: Specifies the node being cloned, adopted, imported, or renamed. This is `null` when the node is being deleted.

`dst` of type `Node`
: Specifies the node newly created if any, or `null`.

No Return Value
No Exceptions

Interface *DOMError* (introduced in **DOM Level 3**)

`DOMError` is an interface that describes an error.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMError {

  // ErrorSeverity
  const unsigned short      SEVERITY_WARNING              = 1;
  const unsigned short      SEVERITY_ERROR                = 2;
  const unsigned short      SEVERITY_FATAL_ERROR          = 3;

  readonly attribute unsigned short  severity;
  readonly attribute DOMString       message;
  readonly attribute DOMString       type;
  readonly attribute DOMObject       relatedException;
  readonly attribute DOMObject       relatedData;
  readonly attribute DOMLocator      location;
};
```

Definition group *ErrorSeverity*

An integer indicating the severity of the error.

Defined Constants

`SEVERITY_ERROR`
: The severity of the error described by the `DOMError` is error. A `SEVERITY_ERROR` may not cause the processing to stop if the error can be recovered, unless `DOMErrorHandler.handleError()` [p.105] returns `false`.

`SEVERITY_FATAL_ERROR`
: The severity of the error described by the `DOMError` is fatal error. A `SEVERITY_FATAL_ERROR` will cause the normal processing to stop. The return value of `DOMErrorHandler.handleError()` [p.105] is ignored unless the implementation chooses to continue, in which case the behavior becomes undefined.

`SEVERITY_WARNING`
: The severity of the error described by the `DOMError` is warning. A `SEVERITY_WARNING` will not cause the processing to stop, unless `DOMErrorHandler.handleError()` [p.105] returns `false`.

Attributes

`location` of type `DOMLocator` [p.106] , readonly
: The location of the error.

`message` of type `DOMString` [p.24] , readonly
: An implementation specific string describing the error that occurred.

`relatedData` of type `DOMObject` [p.25] , readonly
: The related `DOMError.type` [p.105] dependent data if any.

`relatedException` of type `DOMObject` [p.25] , readonly
: The related platform dependent exception if any.

`severity` of type `unsigned short`, readonly
: The severity of the error, either `SEVERITY_WARNING`, `SEVERITY_ERROR`, or `SEVERITY_FATAL_ERROR`.

`type` of type `DOMString` [p.24] , readonly
: A `DOMString` [p.24] indicating which related data is expected in `relatedData`. Users should refer to the specification of the error in order to find its `DOMString` type and `relatedData` definitions if any.

  Note: As an example, `Document.normalizeDocument()` [p.54] does generate warnings when the "split-cdata-sections [p.110] " parameter is in use. Therefore, the method generates a `SEVERITY_WARNING` with `type "cdata-sections-splitted"` and the first `CDATASection` [p.114] node in document order resulting from the split is returned by the `relatedData` attribute.

Interface *DOMErrorHandler* (introduced in **DOM Level 3**)

`DOMErrorHandler` is a callback interface that the DOM implementation can call when reporting errors that happens while processing XML data, or when doing some other processing (e.g. validating a document). A `DOMErrorHandler` object can be attached to a `Document` [p.41] using the "error-handler [p.108] " on the `DOMConfiguration` [p.106] interface. If more than one error needs to be reported during an operation, the sequence and numbers of the errors passed to the error handler are implementation dependent.

The application that is using the DOM implementation is expected to implement this interface.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMErrorHandler {
  boolean            handleError(in DOMError error);
};
```

Methods

`handleError`
: This method is called on the error handler when an error occurs.
  If an exception is thrown from this method, it is considered to be equivalent of returning `true`.

  Parameters

  `error` of type `DOMError` [p.104]
  : The error object that describes the error. This object may be reused by the DOM implementation across multiple calls to the `handleError` method.

  Return Value

  `boolean` If the `handleError` method returns `false`, the DOM implementation should stop the current processing when possible. If the method returns `true`, the processing may continue depending on `DOMError.severity` [p.105] .

No Exceptions

Interface *DOMLocator* (introduced in **DOM Level 3**)

`DOMLocator` is an interface that describes a location (e.g. where an error occurred).

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMLocator {
  readonly attribute long            lineNumber;
  readonly attribute long            columnNumber;
  readonly attribute long            byteOffset;
  readonly attribute long            utf16Offset;
  readonly attribute Node            relatedNode;
  readonly attribute DOMString       uri;
};
```

Attributes

`byteOffset` of type `long`, readonly
: The byte offset into the input source this locator is pointing to or `-1` if there is no byte offset available.

`columnNumber` of type `long`, readonly
: The column number this locator is pointing to, or `-1` if there is no column number available.

`lineNumber` of type `long`, readonly
: The line number this locator is pointing to, or `-1` if there is no column number available.

`relatedNode` of type `Node` [p.56] , readonly
: The node this locator is pointing to, or `null` if no node is available.

`uri` of type `DOMString` [p.24] , readonly
: The URI this locator is pointing to, or `null` if no URI is available.

`utf16Offset` of type `long`, readonly
: The UTF-16, as defined in [*Unicode*] and Amendment 1 of [*ISO/IEC 10646*], offset into the input source this locator is pointing to or `-1` if there is no UTF-16 offset available.

Interface *DOMConfiguration* (introduced in **DOM Level 3**)

The `DOMConfiguration` interface represents the configuration of a document and maintains a table of recognized parameters. Using the configuration, it is possible to change `Document.normalizeDocument()` [p.54] behavior, such as replacing the `CDATASection` [p.114] nodes with `Text` [p.95] nodes or specifying the type of the schema [p.208] that must be used when the validation of the `Document` [p.41] is requested. `DOMConfiguration` objects are also used in [*DOM Level 3 Load and Save*] in the `DOMParser` and `DOMSerializer` interfaces.

The parameter names used by the `DOMConfiguration` object are defined throughout the DOM Level 3 specifications. Names are case-insensitive. To avoid possible conflicts, as a convention, names referring to parameters defined outside the DOM specification should be made unique. Because parameters are exposed as properties in the ECMAScript Language Binding [p.185] , names are recommended to follow the section *"5.16 Identifiers"* of [*Unicode*] with the addition of the character ’-’ (HYPHEN-MINUS) but it is not enforced by the DOM implementation. DOM Level 3 Core Implementations are required to recognize all parameters defined in this specification. Some parameter values may also be required to be supported by the implementation. Refer to the definition

of the parameter to know if a value must be supported or not.

Note: Parameters are similar to features and properties used in SAX2 [*SAX*].

The following list of parameters defined in the DOM:

`"canonical-form"`

`true`

[*optional*]

Canonicalize the document according to the rules specified in [*Canonical XML*], such as removing the `DocumentType` [p.115] node (if any) from the tree, or removing superfluous namespace declarations from each element. Note that this is limited to what can be represented in the DOM; in particular, there is no way to specify the order of the attributes in the DOM. In addition,

Setting this parameter to `true` will also set the state of the parameters listed below. Later changes to the state of one of those parameters will revert "canonical-form [p.107] " back to `false`.

Parameters set to `false`: "entities [p.108] ", "normalize-characters [p.109] ", "cdata-sections [p.107] ".

Parameters set to `true`: "namespaces [p.109] ", "namespace-declarations [p.109] ", "well-formed [p.111] ", "element-content-whitespace [p.108] ".

Other parameters are not changed unless explicitly specified in the description of the parameters.

`false`

[*required*] (*default*)

Do not canonicalize the document.

`"cdata-sections"`

`true`

[*required*] (*default*)

Keep `CDATASection` [p.114] nodes in the document.

`false`

[*required*]

Transform `CDATASection` [p.114] nodes in the document into `Text` [p.95] nodes. The new `Text` node is then combined with any adjacent `Text` node.

`"check-character-normalization"`

`true`

[*optional*]

Check if the characters in the document are fully normalized, as defined in appendix B of [*XML 1.1*]. When a sequence of characters is encountered that fails normalization checking, an error with the `DOMError.type` [p.105] equals to "check-character-normalization-failure" is issued.

`false`

[*required*] (*default*)

Do not check if characters are normalized.

`"comments"`

`true`

[*required*] (*default*)

Keep `Comment` [p.99] nodes in the document.

`false`
[*required*]
Discard `Comment` [p.99] nodes in the document.

`"datatype-normalization"`
`true`
[*optional*]
Expose schema normalized values in the tree, such as XML Schema normalized values in the case of XML Schema. Since this parameter requires to have schema [p.208] information, the "validate [p.110] " parameter will also be set to `true`. Having this parameter activated when "validate" is `false` has no effect and no schema-normalization will happen.

Note: Since the document contains the result of the XML 1.0 processing, this parameter does not apply to attribute value normalization as defined in section 3.3.3 of [*XML 1.0*] and is only meant for schema [p.208] languages other than Document Type Definition (DTD).

`false`
[*required*] (*default*)
Do not perform schema normalization on the tree.

`"element-content-whitespace"`
`true`
[*required*] (*default*)
Keep all whitespaces in the document.
`false`
[*optional*]
Discard all `Text` [p.95] nodes that contain whitespaces in element content, as described in *[element content whitespace]*. The implementation is expected to use the attribute `Text.isElementContentWhitespace` [p.96] to determine if a `Text` node should be discarded or not.

`"entities"`
`true`
[*required*] (*default*)
Keep `EntityReference` [p.118] nodes in the document.
`false`
[*required*]
Remove all `EntityReference` [p.118] nodes from the document, putting the entity expansions directly in their place. `Text` [p.95] nodes are normalized, as defined in `Node.normalize` [p.71] . Only unexpanded entity references are kept in the document.

Note: This parameter does not affect `Entity` [p.116] nodes.

`"error-handler"`
[*required*]
Contains a `DOMErrorHandler` [p.105] object. If an error is encountered in the document, the implementation will call back the `DOMErrorHandler` registered using this parameter. The implementation may provide a default `DOMErrorHandler` object.
When called, `DOMError.relatedData` [p.104] will contain the closest node to where the

error occurred. If the implementation is unable to determine the node where the error occurs, `DOMError.relatedData` will contain the `Document` [p.41] node. Mutations to the document from within an error handler will result in implementation dependent behavior.

`"infoset"`

`true`

[*required*]

Keep in the document the information defined in the XML Information Set [*XML Information Set*].

This forces the following parameters to `false`: "validate-if-schema [p.111] ", "entities [p.108] ", "datatype-normalization [p.108] ", "cdata-sections [p.107] ".

This forces the following parameters to `true`: "namespace-declarations [p.109] ", "well-formed [p.111] ", "element-content-whitespace [p.108] ", "comments [p.107] ", "namespaces [p.109] ".

Other parameters are not changed unless explicitly specified in the description of the parameters.

Note that querying this parameter with `getParameter` returns `true` only if the individual parameters specified above are appropriately set.

`false`

Setting `infoset` to `false` has no effect.

`"namespaces"`

`true`

[*required*] (*default*)

Perform the namespace processing as defined in Namespace Normalization [p.125] .

`false`

[*optional*]

Do not perform the namespace processing.

`"namespace-declarations"`

This parameter has no effect if the parameter "namespaces [p.109] " is set to `false`.

`true`

[*required*] (*default*)

Include namespace declaration attributes, specified or defaulted from the schema [p.208] , in the document. See also the sections "Declaring Namespaces" in [*XML Namespaces*] and [*XML Namespaces 1.1*].

`false`

[*required*]

Discard all namespace declaration attributes. The namespace prefixes (`Node.prefix` [p.62]) are retained even if this parameter is set to `false`.

`"normalize-characters"`

`true`

[*optional*]

Fully normalized the characters in the document as defined in appendix B of [*XML 1.1*].

`false`

[*required*] (*default*)

Do not perform character normalization.

`"schema-location"`

[*optional*]

Represent a `DOMString` [p.24] object containing a list of URIs, separated by whitespaces (characters matching the *nonterminal production S* defined in section 2.3 [*XML 1.0*]), that represents the schemas [p.208] against which validation should occur, i.e. the current schema. The types of schemas referenced in this list must match the type specified with `schema-type`, otherwise the behavior of an implementation is undefined.
The schemas specified using this property take precedence to the schema information specified in the document itself. For namespace aware schema, if a schema specified using this property and a schema specified in the document instance (i.e. using the `schemaLocation` attribute) in a schema document (i.e. using schema `import` mechanisms) share the same `targetNamespace`, the schema specified by the user using this property will be used. If two schemas specified using this property share the same `targetNamespace` or have no namespace, the behavior is implementation dependent.
If no location has been provided, this parameter is `null`.

Note: The `"schema-location"` parameter is ignored unless the "schema-type [p.110] " parameter value is set. It is strongly recommended that `Document.documentURI` [p.42] will be set so that an implementation can successfully resolve any external entities referenced.

`"schema-type"`
[*optional*]
Represent a `DOMString` [p.24] object containing an absolute URI and representing the type of the schema [p.208] language used to validate a document against. Note that no lexical checking is done on the absolute URI.
If this parameter is not set, a default value may be provided by the implementation, based on the schema languages supported and on the schema language used at load time. If no value is provided, this parameter is `null`.

Note: For XML Schema [*XML Schema Part 1*], applications must use the value `"http://www.w3.org/2001/XMLSchema"`. For XML DTD [*XML 1.0*], applications must use the value `"http://www.w3.org/TR/REC-xml"`. Other schema languages are outside the scope of the W3C and therefore should recommend an absolute URI in order to use this method.

`"split-cdata-sections"`
`true`
[*required*] (*default*)
Split CDATA sections containing the CDATA section termination marker ’]]>’. When a CDATA section is split a warning is issued with a `DOMError.type` [p.105] equals to `"cdata-sections-splitted"` and `DOMError.relatedData` [p.104] equals to the first `CDATASection` [p.114] node in document order resulting from the split.
`false`
[*required*]
Signal an error if a `CDATASection` [p.114] contains an unrepresentable character.
`"validate"`
`true`
[*optional*]
Require the validation against a schema [p.208] (i.e. XML schema, DTD, any other type or

representation of schema) of the document as it is being normalized as defined by [*XML 1.0*]. If validation errors are found, or no schema was found, the error handler is notified. Schema-normalized values will not be exposed according to the schema in used unless the parameter "datatype-normalization [p.108] " is `true`.
This parameter will reevaluate:

- Attribute nodes with `Attr.specified` [p.84] equals to `false`, as specified in the description of the `Attr` [p.81] interface;
- The value of the attribute `Text.isElementContentWhitespace` [p.96] for all `Text` [p.95] nodes;
- The value of the attribute `Attr.isId` [p.83] for all `Attr` [p.81] nodes;
- The attributes `Element.schemaTypeInfo` [p.86] and `Attr.schemaTypeInfo` [p.84] .

Note: "validate-if-schema [p.111] " and "validate" are mutually exclusive, setting one of them to `true` will set the other one to `false`. Applications should also consider setting the parameter "well-formed [p.111] " to `true`, which is the default for that option, when validating the document.

`false`
[*required*] (*default*)
Do not accomplish schema processing, including the internal subset processing. Default attribute values information are kept. Note that validation might still happen if "validate-if-schema [p.111] " is `true`.

`"validate-if-schema"`
`true`
[*optional*]
Enable validation only if a declaration for the document element can be found in a schema [p.208] (independently of where it is found, i.e. XML schema, DTD, or any other type or representation of schema). If validation is enabled, this parameter has the same behavior as the parameter "validate [p.110] " set to `true`.

Note: "validate-if-schema" and "validate [p.110] " are mutually exclusive, setting one of them to `true` will set the other one to `false`.

`false`
[*required*] (*default*)
No schema processing should be performed if the document has a schema, including internal subset processing. Default attribute values information are kept. Note that validation must still happen if "validate [p.110] " is `true`.

`"well-formed"`
`true`
[*required*] (*default*)
Check if all nodes are XML well formed [p.208] according to the XML version in use in `Document.xmlVersion` [p.43] :

- check if the attribute `Node.nodeName` [p.62] contains invalid characters according to its node type and generate a `DOMError` [p.104] of type `"wf-invalid-character-in-node-name"`, with a

`DOMError.SEVERITY_ERROR` [p.104] severity, if necessary;

- check if the text content inside `Attr` [p.81] , `Element` [p.85] , `Comment` [p.99] , `Text` [p.95] , `CDATASection` [p.114] nodes for invalid characters and generate a `DOMError` [p.104] of type `"wf-invalid-character"`, with a `DOMError.SEVERITY_ERROR` [p.104] severity, if necessary;
- check if the data inside `ProcessingInstruction` [p.118] nodes for invalid characters and generate a `DOMError` [p.104] of type `"wf-invalid-character"`, with a `DOMError.SEVERITY_ERROR` [p.104] severity, if necessary;

`false`

[*optional*]
Do not check for XML well-formedness.

The resolution of the system identifiers associated with entities is done using `Document.documentURI` [p.42] . However, when the feature "LS" defined in [*DOM Level 3 Load and Save*] is supported by the DOM implementation, the parameter "resource-resolver" can also be used on `DOMConfiguration` objects attached to `Document` [p.41] nodes. If this parameter is set, `Document.normalizeDocument()` [p.54] will invoke the resource resolver instead of using `Document.documentURI`.

IDL Definition

```
// Introduced in DOM Level 3:
interface DOMConfiguration {
  void               setParameter(in DOMString name,
                                  in DOMUserData value)
                                     raises(DOMException);
  DOMUserData        getParameter(in DOMString name)
                                     raises(DOMException);
  boolean            canSetParameter(in DOMString name,
                                     in DOMUserData value);
  readonly attribute DOMStringList   parameterNames;
};
```

Attributes

`parameterNames` of type `DOMStringList` [p.33] , readonly

The list of the parameters supported by this `DOMConfiguration` object and for which at least one value can be set by the application. Note that this list can also contain parameter names defined outside this specification.

Methods

`canSetParameter`

Check if setting a parameter to a specific value is supported.

Parameters

`name` of type `DOMString` [p.24]

The name of the parameter to check.

`value` of type `DOMUserData` [p.25]

An object. if `null`, the returned value is `true`.

Return Value

`boolean`	`true` if the parameter could be successfully set to the specified value, or `false` if the parameter is not recognized or the requested value is not supported. This does not change the current value of the parameter itself.

No Exceptions

`getParameter`

Return the value of a parameter if known.

Parameters

`name` of type `DOMString` [p.24]

The name of the parameter.

Return Value

`DOMUserData` [p.25]	The current object associated with the specified parameter or `null` if no object has been associated or if the parameter is not supported.

Exceptions

`DOMException` [p.31]	NOT_FOUND_ERR: Raised when the parameter name is not recognized.

`setParameter`

Set the value of a parameter.

Parameters

`name` of type `DOMString` [p.24]

The name of the parameter to set.

`value` of type `DOMUserData` [p.25]

The new value or `null` if the user wishes to unset the parameter. While the type of the value parameter is defined as `DOMUserData`, the object type must match the type defined by the definition of the parameter. For example, if the parameter is "error-handler" [p.108] , the value must be of type `DOMErrorHandler` [p.105] .

Exceptions

`DOMException` [p.31]	NOT_FOUND_ERR: Raised when the parameter name is not recognized. NOT_SUPPORTED_ERR: Raised when the parameter name is recognized but the requested value cannot be set. TYPE_MISMATCH_ERR: Raised if the value type for this parameter name is incompatible with the expected value type.

No Return Value

1.5 Extended Interfaces: XML Module

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.

The interfaces found within this section are not mandatory. A DOM application may use the `DOMImplementation.hasFeature(feature, version)` [p.40] method with parameter values "XML" and "3.0" (respectively) to determine whether or not this module is supported by the implementation. In order to fully support this module, an implementation must also support the "Core" feature defined in Fundamental Interfaces: Core Module [p.30] and the feature "XMLVersion" with version "1.0" defined in `Document.xmlVersion` [p.43] . Please refer to additional information about Conformance [p.17] in this specification. The DOM Level 3 XML module is backward compatible with the DOM Level 2 XML [*DOM Level 2 Core*] and DOM Level 1 XML [*DOM Level 1*] modules, i.e. a DOM Level 3 XML implementation who returns `true` for "XML" with the `version` number `"3.0"` must also return `true` for this `feature` when the `version` number is `"2.0"`, `"1.0"`, `""` or, `null`.

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 `CharacterData.data` [p.79] attribute 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.78] interface through the `Text` [p.95] interface. Adjacent `CDATASection` nodes are not merged by use of the `normalize` method of the `Node` [p.56] interface.

No lexical check is done on the content of a CDATA section and it is therefore possible to have the character sequence `"]]>"` in the content, which is illegal in a CDATA section per section 2.7 of [*XML 1.0*]. The presence of this character sequence must generate a fatal error during serialization or the cdata section must be splitted before the serialization (see also the parameter `"split-cdata-sections"` in the `DOMConfiguration` [p.106] 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.41] 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.

DOM Level 3 doesn't support editing `DocumentType` nodes. `DocumentType` nodes are read-only [p.207] .

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.73] , readonly

A `NamedNodeMap` [p.73] 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.116] interface.

The DOM Level 2 does not support editing entities, therefore `entities` cannot be altered in any way.

`internalSubset` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 2**

The internal subset as a string, or `null` if there is none. This is does not contain the delimiting square brackets.

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.24] , readonly
: The name of DTD; i.e., the name immediately following the `DOCTYPE` keyword.

`notations` of type `NamedNodeMap` [p.73] , readonly
: A `NamedNodeMap` [p.73] containing the notations declared in the DTD. Duplicates are discarded. Every node in this map also implements the `Notation` [p.116] interface. The DOM Level 2 does not support editing notations, therefore `notations` cannot be altered in any way.

`publicId` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 2**
: The public identifier of the external subset.

`systemId` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 2**
: The system identifier of the external subset. This may be an absolute URI or not.

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 1.0*]), or is used for formal declaration of processing instruction targets (see *section 2.6* of the XML 1.0 specification [*XML 1.0*]). The `nodeName` attribute inherited from `Node` [p.56] is set to the declared name of the notation.

The DOM Core does not support editing `Notation` nodes; they are therefore readonly [p.207] .

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.24] , readonly
: The public identifier of this notation. If the public identifier was not specified, this is `null`.

`systemId` of type `DOMString` [p.24] , readonly
: The system identifier of this notation. If the system identifier was not specified, this is `null`. This may be an absolute URI or not.

Interface *Entity*

This interface represents a known entity, either parsed or unparsed, in an XML document. Note that this models the entity itself *not* the entity declaration.

The `nodeName` attribute that is inherited from `Node` [p.56] 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.118] 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 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 text of the entity may not be available. When the *replacement text* is available, the corresponding `Entity` node's child list represents the structure of that replacement value. Otherwise, the child list is empty.

DOM Level 3 does not support editing `Entity` nodes; if a user wants to make changes to the contents of an `Entity`, every related `EntityReference` [p.118] 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.205] are readonly [p.207] .

An `Entity` node does not have any parent.

Note: If the entity contains an unbound namespace prefix [p.207] , the `namespaceURI` of the corresponding node in the `Entity` node subtree is `null`. The same is true for `EntityReference` [p.118] nodes that refer to this entity, when they are created using the `createEntityReference` method of the `Document` [p.41] interface.

IDL Definition

```
interface Entity : Node {
  readonly attribute DOMString       publicId;
  readonly attribute DOMString       systemId;
  readonly attribute DOMString       notationName;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       inputEncoding;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       xmlEncoding;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       xmlVersion;
};
```

Attributes

`inputEncoding` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
: An attribute specifying the encoding used for this entity at the time of parsing, when it is an external parsed entity. This is `null` if it an entity from the internal subset or if it is not known.

`notationName` of type `DOMString` [p.24] , readonly
: For unparsed entities, the name of the notation for the entity. For parsed entities, this is `null`.

`publicId` of type `DOMString` [p.24] , readonly
: The public identifier associated with the entity if specified, and `null` otherwise.

`systemId` of type `DOMString` [p.24] , readonly
: The system identifier associated with the entity if specified, and `null` otherwise. This may be an absolute URI or not.

`xmlEncoding` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
: An attribute specifying, as part of the text declaration, the encoding of this entity, when it is an external parsed entity. This is `null` otherwise.

`xmlVersion` of type `DOMString` [p.24] , readonly, introduced in **DOM Level 3**
: An attribute specifying, as part of the text declaration, the version number of this entity, when it is an external parsed entity. This is `null` otherwise.

Interface *EntityReference*

`EntityReference` nodes may be used to represent an entity reference in the tree. 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 `Document` [p.41] , instead of providing `EntityReference` nodes. If it does provide such nodes, then for an `EntityReference` node that represents a reference to a known entity an `Entity` [p.116] exists, and the subtree of the `EntityReference` node is a copy of the `Entity` node subtree. However, the latter may not be true when an entity contains an unbound namespace prefix [p.207] . In such a case, because the namespace prefix resolution depends on where the entity reference is, the descendants [p.205] of the `EntityReference` node may be bound to different namespace URIs [p.207] . When an `EntityReference` node represents a reference to an unknown entity, the node has no children and its replacement value, when used by `Attr.value` [p.84] for example, is empty.

As for `Entity` [p.116] nodes, `EntityReference` nodes and all their descendants [p.205] are readonly [p.207] .

Note: `EntityReference` nodes may cause element content and attribute value normalization problems when, such as in XML 1.0 and XML Schema, the normalization is performed after entity reference are expanded.

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.

No lexical check is done on the content of a processing instruction and it is therefore possible to have the character sequence `"?>"` in the content, which is illegal a processing instruction per section 2.6 of [*XML 1.0*]. The presence of this character sequence must generate a fatal error during serialization.

IDL Definition

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

};
```

Attributes

`data` of type `DOMString` [p.24]

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.31]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

`target` of type `DOMString` [p.24] , readonly

The target of this processing instruction. XML defines this as being the first token [p.208] following the markup that begins the processing instruction.

Appendix A: Changes

Editor:
Philippe Le Hégaret, W3C

This section summarizes the changes between [*DOM Level 2 Core*] and this new version of the Core specification.

A.1 New sections

The following new sections have been added:

- DOM Architecture [p.16] : a global overview of the DOM Level 3 modules;
- DOM URIs [p.26] : general considerations on the URI handling in DOM Level 3;
- Base URIs [p.28] : How the **[base URI]** property defined in [*XML Information Set*] has been exposed in DOM Level 3;
- Mixed DOM Implementations [p.28] : general considerations on DOM implementation extensions;
- DOM Features [p.29] : overview of the DOM features and how they relate to the DOM modules;
- Bootstrapping [p.30] : general introduction to the DOM Level 3 bootstrapping mechanisms;
- Namespaces Algorithms [p.125] : how namespace URIs and prefixes are resolved in DOM Level 3;
- Infoset Mapping [p.147] : relation between DOM Level 3 and [*XML Information Set*];
- Configuration Settings [p.145] : relations between parameters as used in `DOMConfiguration` [p.106] ;

A.2 Changes to DOM Level 2 Core interfaces and exceptions

Interface `Attr` [p.81]
: The `Attr` [p.81] interface has two new attributes, `Attr.schemaTypeInfo` [p.84] , and `Attr.isId` [p.83] .

Interface `Document` [p.41]
: The `Document` [p.41] interface has seven new attributes: `Document.inputEncoding` [p.43] , `Document.xmlEncoding` [p.43] , `Document.xmlStandalone` [p.43] , `Document.xmlVersion` [p.43] , `Document.strictErrorChecking` [p.43] , `Document.documentURI` [p.42] , and `Document.domConfig` [p.43] . It has three new methods: `Document.adoptNode(source)` [p.44] , `Document.normalizeDocument()` [p.54] , and `Document.renameNode(n, namespaceURI, qualifiedName)` [p.55] . The attribute `Document.doctype` [p.42] has been modified.

Exception `DOMException` [p.31]
: The `DOMException` [p.31] has two new exception codes: `VALIDATION_ERR` [p.33] and `TYPE_MISMATCH_ERR` [p.32] .

Interface `DOMImplementation` [p.37]
: The `DOMImplementation` [p.37] interface has one new method, `DOMImplementation.getFeature(feature, version)` [p.39] .

Interface `Entity` [p.116]
: The `Entity` [p.116] interface has three new attributes: `Entity.inputEncoding` [p.117] , `Entity.xmlEncoding` [p.118] , and `Entity.xmlVersion` [p.118] .

Interface `Element` [p.85]
: The `Element` [p.85] interface has one new attribute, `Element.schemaTypeInfo` [p.86] , and three new methods: `Element.setIdAttribute(name, isId)` [p.94] , `Element.setIdAttributeNS(namespaceURI, localName, isId)` [p.94] , and `Element.setIdAttributeNode(idAttr, isId)` [p.95] .

Interface `Node` [p.56]
: The `Node` [p.56] interface has two new attributes, `Node.baseURI` [p.61] and `Node.textContent` [p.63] . It has nine new methods: `Node.compareDocumentPosition(other)` [p.66] , `Node.isSameNode(other)` [p.69] , `Node.lookupPrefix(namespaceURI)` [p.70] , `Node.isDefaultNamespace(namespaceURI)` [p.68] , `Node.lookupNamespaceURI(prefix)` [p.70] , `Node.isEqualNode(arg)` [p.68] , `Node.getFeature(feature, version)` [p.66] , `Node.setUserData(key, data, handler)` [p.72] , `Node.getUserData(key)` [p.67] . It introduced 6 new constants: `Node.DOCUMENT_POSITION_DISCONNECTED` [p.61] , `Node.DOCUMENT_POSITION_PRECEDING` [p.61] , `Node.DOCUMENT_POSITION_FOLLOWING` [p.61] , `Node.DOCUMENT_POSITION_CONTAINS` [p.60] , `Node.DOCUMENT_POSITION_CONTAINED_BY` [p.60] , and `Node.DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC` [p.61] . The methods `Node.insertBefore(newChild, refChild)` [p.67] , `Node.replaceChild(newChild, oldChild)` [p.71] and `Node.removeChild(oldChild)` [p.71] have been modified.

Interface `Text` [p.95]
: The `Text` [p.95] interface has two new attributes, `Text.wholeText` [p.96] and `Text.isElementContentWhitespace` [p.96] , and one new method, `Text.replaceWholeText(content)` [p.97] .

A.3 New DOM features

"XMLVersion"
: The "XMLVersion" DOM feature was introduced to represent if an implementation is able to support [*XML 1.0*] or [*XML 1.1*]. See `Document.xmlVersion` [p.43] .

A.4 New types

`DOMUserData` [p.25]
: The `DOMUserData` [p.25] type was added to the Core module.

`DOMObject` [p.25]
: The `DOMObject` [p.25] type was added to the Core module.

A.5 New interfaces

`DOMStringList` [p.33]
: The `DOMStringList` [p.33] interface has one attribute, `DOMStringList.length` [p.33] , and one method, `DOMStringList.item(index)` [p.33] .

`NameList` [p.34]
: The `NameList` [p.34] interface has one attribute, `NameList.length` [p.34] , and two methods, `NameList.getName(index)` [p.35] and `NameList.getNamespaceURI(index)` [p.35] .

`DOMImplementationList` [p.35]
: The `DOMImplementationList` [p.35] interface has one attribute, `DOMImplementationList.length` [p.35] , and one method, `DOMImplementationList.item(index)` [p.35] .

`DOMImplementationSource` [p.36]
: The `DOMImplementationSource` [p.36] interface has two methods, `DOMImplementationSource.getDOMImplementation(features)` [p.36] , and `DOMImplementationSource.getDOMImplementationList(features)` [p.37] .

`TypeInfo` [p.99]
: The `TypeInfo` [p.99] interface has two attributes, `TypeInfo.typeName` [p.102] , and `TypeInfo.typeNamespace` [p.102] .

`UserDataHandler` [p.102]
: The `UserDataHandler` [p.102] interface has one method, `UserDataHandler.handle(operation, key, data, src, dst)` [p.103] , and four constants: `UserDataHandler.NODE_CLONED` [p.103] , `UserDataHandler.NODE_IMPORTED` [p.103] , `UserDataHandler.NODE_DELETED` [p.103] , and `UserDataHandler.NODE_RENAMED` [p.103] .

`DOMError` [p.104]
: The `DOMError` [p.104] interface has six attributes: `DOMError.severity` [p.105] , `DOMError.message` [p.104] , `DOMError.type` [p.105] , `DOMError.relatedException` [p.104] , `DOMError.relatedData` [p.104] , and `DOMError.location` [p.104] . It has four constants: `DOMError.SEVERITY_WARNING` [p.104] , `DOMError.SEVERITY_ERROR` [p.104] , and `DOMError.SEVERITY_FATAL_ERROR` [p.104] .

`DOMErrorHandler` [p.105]
: The `DOMErrorHandler` [p.105] interface has one method: `DOMErrorHandler.handleError(error)` [p.105] .

`DOMLocator` [p.106]
: The `DOMLocator` [p.106] interface has seven attributes: `DOMLocator.lineNumber` [p.106] , `DOMLocator.columnNumber` [p.106] , `DOMLocator.byteOffset` [p.106] , `DOMLocator.utf16Offset` [p.106] , `DOMLocator.relatedNode` [p.106] , `DOMLocator.uri` [p.106] , and `DOMLocator.lineNumber`.

`DOMConfiguration` [p.106]
: The `DOMConfiguration` [p.106] interface has one attribute: `DOMConfiguration.parameterNames` [p.112] . It also has three methods: `DOMConfiguration.setParameter(name, value)` [p.113] , `DOMConfiguration.getParameter(name)` [p.113] , and `DOMConfiguration.canSetParameter(name, value)` [p.112] .

A.6 Objects

This specification defines one object, only provided in the bindings:

`DOMImplementationRegistry`
: The `DOMImplementationRegistry` object has two methods, `DOMImplementationRegistry.getDOMImplementation(features)`, and `DOMImplementationRegistry.getDOMImplementationList(features)`.

Appendix B: Namespaces Algorithms

Editors:
Arnaud Le Hors, IBM
Elena Litani, IBM

This appendix contains several namespace algorithms, such as namespace normalization algorithm that fixes namespace information in the Document Object Model to produce a namespace well-formed [p.207] document. If [*XML 1.0*] is in use (see `Document.xmlVersion` [p.43]) the algorithms conform to [*XML Namespaces*], otherwise if [*XML 1.1*] is in use, algorithms conform to [*XML Namespaces 1.1*].

B.1 Namespace Normalization

Namespace declaration attributes and prefixes are normalized as part of the `normalizeDocument` method of the `Document` [p.41] interface as if the following method described in pseudo code was called on the document element.

```
void Element.normalizeNamespaces()
{

  // Pick up local namespace declarations
  //
  for ( all DOM Level 2 valid local namespace declaration attributes of Element )
  {
      if (the namespace declaration is invalid)
      {
          // Note: The prefix xmlns is used only to declare namespace bindings and
          // is by definition bound to the namespace name http://www.w3.org/2000/xmlns/.
          // It must not be declared. No other prefix may be bound to this namespace name.

          ==> Report an error.

      }
      else
      {
          ==>  Record the namespace declaration
      }
  }


  // Fixup element's namespace
  //
  if ( Element's namespaceURI != null )
  {
    if ( Element's prefix/namespace pair (or default namespace,
         if no prefix) are within the scope of a binding )
    {
      ==> do nothing, declaration in scope is inherited

      See section "B.1.1: Scope of a binding" for an example

    }
    else
    {
      ==> Create a local namespace declaration attr for this namespace,
```

```
        with Element's current prefix (or a default namespace, if
        no prefix). If there's a conflicting local declaration
        already present, change its value to use this namespace.

        See section "B.1.2: Conflicting namespace declaration" for an example

        // NOTE that this may break other nodes within this Element's
        // subtree, if they're already using this prefix.
        // They will be repaired when we reach them.
  }
}
else
{
  // Element has no namespace URI:
  if ( Element's localName is null )
  {
     // DOM Level 1 node
     ==> if in process of validation against a namespace aware schema
         (i.e XML Schema) report a fatal error: the processor can not recover
         in this situation.
         Otherwise, report an error: no namespace fixup will be performed on this node.
  }
  else
  {
    // Element has no pseudo-prefix
    if ( there's a conflicting local default namespace declaration
         already present )
    {
      ==> change its value to use this empty namespace.

    }
    // NOTE that this may break other nodes within this Element's
    // subtree, if they're already using the default namespaces.
    // They will be repaired when we reach them.
  }
}


// Examine and polish the attributes
//
for ( all non-namespace Attrs of Element )
{
   if ( Attr[i] has a namespace URI )
   {
      if ( attribute has no prefix (default namespace decl does not apply to attributes)
           OR
           attribute prefix is not declared
           OR
           conflict: attribute has a prefix that conflicts with a binding
                     already active in scope)
      {
         if (namespaceURI matches an in scope declaration of one or more prefixes)
         {
             // pick the most local binding available;
             // if there is more than one pick one arbitrarily

             ==> change attribute's prefix.
         }
         else
         {
             if (the current prefix is not null and it has no in scope declaration)
```

```
                {
                    ==> declare this prefix
                }
                else
                {
                    // find a prefix following the pattern "NS" +index (starting at 1)
                    // make sure this prefix is not declared in the current scope.
                    // create a local namespace declaration attribute

                    ==> change attribute's prefix.
                }
            }
        }
    }
    else
    {
        // Attr[i] has no namespace URI

        if ( Attr[i] has no localName )
        {
            // DOM Level 1 node
            ==> if in process of validation against a namespace aware schema
                (i.e XML Schema) report a fatal error: the processor can not recover
                 in this situation.
                 Otherwise, report an error: no namespace fixup will be performed on this node.
        }
        else
        {
            // attr has no namespace URI and no prefix
            // no action is required, since attrs don't use default
            ==> do nothing
        }
    }
  } // end for-all-Attrs

  // do this recursively
  for ( all child elements of Element )
  {
    childElement.normalizeNamespaces()
  }
} // end Element.normalizeNamespaces
```

B.1.1 Scope of a Binding

Note: This section is informative.

An element's prefix/namespace URI pair is said to be within the scope of a binding if its namespace prefix is bound to the same namespace URI in the [in-scope namespaces] defined in [*XML Information Set*].

As an example, the following document is loaded in a DOM tree:

```
<root>
  <parent xmlns:ns="http://www.example.org/ns1"
          xmlns:bar="http://www.example.org/ns2">
    <ns:child1 xmlns:ns="http://www.example.org/ns2"/>
  </parent>
</root>
```

In the case of the `child1` element, the namespace prefix and namespace URI are within the scope of the appropriate namespace declaration given that the namespace prefix `ns` of `child1` is bound to `http://www.example.org/ns2`.

Using the method `Node.appendChild` [p.64] , a `child2` element is added as a sibling of `child1` with the same namespace prefix and namespace URI, i.e. `"ns"` and `"http://www.example.org/ns2"` respectively. Unlike `child1` which contains the appropriate namespace declaration in its attributes, `child2`'s prefix/namespace URI pair is within the scope of the namespace declaration of its parent, and the namespace prefix `"ns"` is bound to `"http://www.example.org/ns1"`. `child2`'s prefix/namespace URI pair is therefore not within the scope of a binding. In order to put them within a scope of a binding, the namespace normalization algorithm will create a namespace declaration attribute value to bind the namespace prefix `"ns"` to the namespace URI `"http://www.example.org/ns2"` and will attach to `child2`. The XML representation of the document after the completion of the namespace normalization algorithm will be:

```
<root>
  <parent xmlns:ns="http://www.example.org/ns1"
          xmlns:bar="http://www.example.org/ns2">
    <ns:child1 xmlns:ns="http://www.example.org/ns2"/>
    <ns:child2 xmlns:ns="http://www.example.org/ns2"/>
  </parent>
</root>
```

To determine if an element is within the scope of a binding, one can invoke `Node.lookupNamespaceURI` [p.70] , using its namespace prefix as the parameter, and compare the resulting namespace URI against the desired URI, or one can invoke `Node.isDefaultNamespaceURI` using its namespace URI if the element has no namespace prefix.

B.1.2 Conflicting Namespace Declaration

Note: This section is informative.

A conflicting namespace declaration could occur on an element if an `Element` [p.85] node and a namespace declaration attribute use the same prefix but map them to two different namespace URIs.

As an example, the following document is loaded in a DOM tree:

```
<root>
  <ns:child1 xmlns:ns="http://www.example.org/ns1">
    <ns:child2/>
  </ns:child1>
</root>
```

Using the method `Node.renameNode`, the namespace URI of the element `child1` is renamed from `"http://www.example.org/ns1"` to `"http://www.example.org/ns2"`. The namespace prefix `"ns"` is now mapped to two different namespace URIs at the element `child1` level and thus the namespace declaration is declared conflicting. The namespace normalization algorithm will resolved the namespace prefix conflict by modifying the namespace declaration attribute value from `"http://www.example.org/ns1"` to `"http://www.example.org/ns2"`. The algorithm will then continue and consider the element `child2`, will no longer find a namespace declaration

mapping the namespace prefix `"ns"` to `"http://www.example.org/ns1"` in the element's scope, and will create a new one. The XML representation of the document after the completion of the namespace normalization algorithm will be:

```
<root>
  <ns:child1 xmlns:ns="http://www.example.org/ns2">
    <ns:child2  xmlns:ns="http://www.example.org/ns1"/>
  </ns:child1>
</root>
```

B.2 Namespace Prefix Lookup

The following describes in pseudo code the algorithm used in the `lookupPrefix` method of the `Node` [p.56] interface. Before returning found prefix the algorithm needs to make sure that the prefix is not redefined on an element from which the lookup started. This methods ignores DOM Level 1 nodes.

Note: This method ignores all default namespace declarations. To look up default namespace use `isDefaultNamespace` method.

```
DOMString lookupPrefix(in DOMString namespaceURI)
{
  if (namespaceURI has no value, i.e. namespaceURI is null or empty string) {
     return null;
  }
  short type = this.getNodeType();
  switch (type) {
        case Node.ELEMENT_NODE:
        {
             return lookupNamespacePrefix(namespaceURI, this);
        }
        case Node.DOCUMENT_NODE:
        {
             return getDocumentElement().lookupNamespacePrefix(namespaceURI);
        }
        case Node.ENTITY_NODE :
        case Node.NOTATION_NODE:
        case Node.DOCUMENT_FRAGMENT_NODE:
        case Node.DOCUMENT_TYPE_NODE:
            return null;  // type is unknown
        case Node.ATTRIBUTE_NODE:
        {
             if ( Attr has an owner Element )
             {
                 return ownerElement.lookupNamespacePrefix(namespaceURI);
             }
             return null;
        }
        default:
        {
           if (Node has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
           {
                    return ancestor.lookupNamespacePrefix(namespaceURI);
           }
            return null;
        }
```

```
    }
 }


DOMString lookupNamespacePrefix(DOMString namespaceURI, Element originalElement){
        if ( Element has a namespace and Element's namespace == namespaceURI and
             Element has a prefix and
             originalElement.lookupNamespaceURI(Element's prefix) == namespaceURI)
        {
             return (Element's prefix);
        }
        if ( Element has attributes)
        {
            for ( all DOM Level 2 valid local namespace declaration attributes of Element )
            {
                if (Attr's prefix == "xmlns" and
                   Attr's value == namespaceURI and
                   originalElement.lookupNamespaceURI(Attr's localname) == namespaceURI)
                   {
                      return (Attr's localname);
                   }
            }
        }

        if (Node has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
        {
            return ancestor.lookupNamespacePrefix(namespaceURI, originalElement);
        }
        return null;
    }
```

B.3 Default Namespace Lookup

The following describes in pseudo code the algorithm used in the `isDefaultNamespace` method of the `Node` [p.56] interface. This methods ignores DOM Level 1 nodes.

```
boolean isDefaultNamespace(in DOMString namespaceURI)
{
  switch (nodeType) {
  case ELEMENT_NODE:
     if ( Element has no prefix )
     {
          return (Element's namespace == namespaceURI);
     }
     if ( Element has attributes and there is a valid DOM Level 2
          default namespace declaration, i.e. Attr's localName == "xmlns" )
     {
          return (Attr's value == namespaceURI);
     }

     if ( Element has an ancestor Element )
         // EntityReferences may have to be skipped to get to it
     {
          return ancestorElement.isDefaultNamespace(namespaceURI);
     }
     else {
```

```
          return unknown (false);
      }
  case DOCUMENT_NODE:
      return documentElement.isDefaultNamespace(namespaceURI);
  case ENTITY_NODE:
  case NOTATION_NODE:
  case DOCUMENT_TYPE_NODE:
  case DOCUMENT_FRAGMENT_NODE:
      return unknown (false);
  case ATTRIBUTE_NODE:
      if ( Attr has an owner Element )
      {
          return ownerElement.isDefaultNamespace(namespaceURI);
      }
      else {
          return unknown (false);
      }
  default:
      if ( Node has an ancestor Element )
          // EntityReferences may have to be skipped to get to it
      {
          return ancestorElement.isDefaultNamespace(namespaceURI);
      }
      else {
          return unknown (false);
      }
  }
}
```

B.4 Namespace URI Lookup

The following describes in pseudo code the algorithm used in the `lookupNamespaceURI` method of the `Node` [p.56] interface. This methods ignores DOM Level 1 nodes.

```
DOMString lookupNamespaceURI(in DOMString prefix)
{
  switch (nodeType) {
     case ELEMENT_NODE:
     {
         if ( Element's namespace != null and Element's prefix == prefix )
         {
               // Note: prefix could be "null" in this case we are looking for default namespace
               return (Element's namespace);
         }
         if ( Element has attributes)
         {
            for ( all DOM Level 2 valid local namespace declaration attributes of Element )
            {
                 if (Attr's prefix == "xmlns" and Attr's localName == prefix )
                       // non default namespace
                 {
                        if (Attr's value is not empty)
                        {
                          return (Attr's value);
                        }
                        return unknown (null);
                 }
                 else if (Attr's localname == "xmlns" and prefix == null)
```

```
                    // default namespace
                {
                     if (Attr's value is not empty)
                     {
                       return (Attr's value);
                     }
                     return unknown (null);
                }
            }
        }
        if ( Element has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
        {
                  return ancestorElement.lookupNamespaceURI(prefix);
        }
        return null;
    }
    case DOCUMENT_NODE:
         return documentElement.lookupNamespaceURI(prefix)

    case ENTITY_NODE:
    case NOTATION_NODE:
    case DOCUMENT_TYPE_NODE:
    case DOCUMENT_FRAGMENT_NODE:
          return unknown (null);

    case ATTRIBUTE_NODE:
        if (Attr has an owner Element)
        {
            return ownerElement.lookupNamespaceURI(prefix);
        }
        else
        {
            return unknown (null);
        }
    default:
        if (Node has an ancestor Element)
         // EntityReferences may have to be skipped to get to it
        {
            return ancestorElement.lookupNamespaceURI(prefix);
        }
        else {
            return unknown (null);
        }
  }
}
```

Appendix E: Accessing code point boundaries

Mark Davis, IBM
Lauren Wood, SoftQuad Software Inc.

E.1 Introduction

This appendix is an informative, not a normative, part of the Level 3 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 1.0*] (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.24] 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.

E.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 round-trip from a UTF-32 offset to a UTF-16 offset and back. You can round-trip 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 round-trip from a UTF-32 offset to a UTF-16 offset and back. You can round-trip 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 F: IDL Definitions

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

The IDL files are also available as:
http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/idl.zip

dom.idl:

```
// File: dom.idl

#ifndef _DOM_IDL_
#define _DOM_IDL_

#pragma prefix "w3c.org"
module dom
{

  valuetype DOMString sequence<unsigned short>;

  typedef   unsigned long long DOMTimeStamp;

  typedef   any DOMUserData;

  typedef   Object DOMObject;

  interface DOMImplementation;
  interface DocumentType;
  interface Document;
  interface NodeList;
  interface NamedNodeMap;
  interface UserDataHandler;
  interface Element;
  interface TypeInfo;
  interface DOMLocator;

  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;
// Introduced in DOM Level 3:
const unsigned short      VALIDATION_ERR                = 16;
// Introduced in DOM Level 3:
const unsigned short      TYPE_MISMATCH_ERR             = 17;


// Introduced in DOM Level 3:
interface DOMStringList {
  DOMString          item(in unsigned long index);
  readonly attribute unsigned long   length;
  boolean            contains(in DOMString str);
};

// Introduced in DOM Level 3:
interface NameList {
  DOMString          getName(in unsigned long index);
  DOMString          getNamespaceURI(in unsigned long index);
  readonly attribute unsigned long   length;
  boolean            contains(in DOMString str);
  boolean            containsNS(in DOMString namespaceURI,
                                in DOMString name);
};

// Introduced in DOM Level 3:
interface DOMImplementationList {
  DOMImplementation  item(in unsigned long index);
  readonly attribute unsigned long   length;
};

// Introduced in DOM Level 3:
interface DOMImplementationSource {
  DOMImplementation  getDOMImplementation(in DOMString features);
  DOMImplementationList getDOMImplementationList(in DOMString features);
};

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);
  // Introduced in DOM Level 3:
  DOMObject          getFeature(in DOMString feature,
```

```
                                in DOMString version);
};

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;
  // Modified in DOM Level 3:
  Node               insertBefore(in Node newChild,
                                  in Node refChild)
                                      raises(DOMException);
  // Modified in DOM Level 3:
  Node               replaceChild(in Node newChild,
                                  in Node oldChild)
                                      raises(DOMException);
  // Modified in DOM Level 3:
  Node               removeChild(in Node oldChild)
                                      raises(DOMException);
  // Modified in DOM Level 3:
  Node               appendChild(in Node newChild)
                                      raises(DOMException);
  boolean            hasChildNodes();
  Node               cloneNode(in boolean deep);
  // Modified in DOM Level 3:
  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();
  // Introduced in DOM Level 3:
  readonly attribute DOMString       baseURI;

  // DocumentPosition
  const unsigned short      DOCUMENT_POSITION_DISCONNECTED = 0x01;
  const unsigned short      DOCUMENT_POSITION_PRECEDING    = 0x02;
  const unsigned short      DOCUMENT_POSITION_FOLLOWING    = 0x04;
  const unsigned short      DOCUMENT_POSITION_CONTAINS     = 0x08;
  const unsigned short      DOCUMENT_POSITION_CONTAINED_BY = 0x10;
  const unsigned short      DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;

  // Introduced in DOM Level 3:
  unsigned short     compareDocumentPosition(in Node other)
                                      raises(DOMException);
  // Introduced in DOM Level 3:
           attribute DOMString       textContent;
                                      // raises(DOMException) on setting
                                      // raises(DOMException) on retrieval

  // Introduced in DOM Level 3:
  boolean            isSameNode(in Node other);
  // Introduced in DOM Level 3:
  DOMString          lookupPrefix(in DOMString namespaceURI);
  // Introduced in DOM Level 3:
  boolean            isDefaultNamespace(in DOMString namespaceURI);
  // Introduced in DOM Level 3:
  DOMString          lookupNamespaceURI(in DOMString prefix);
  // Introduced in DOM Level 3:
  boolean            isEqualNode(in Node arg);
  // Introduced in DOM Level 3:
  DOMObject          getFeature(in DOMString feature,
                                in DOMString version);
  // Introduced in DOM Level 3:
  DOMUserData        setUserData(in DOMString key,
                                 in DOMUserData data,
                                 in UserDataHandler handler);
  // Introduced in DOM Level 3:
  DOMUserData        getUserData(in DOMString key);
};

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)
                                     raises(DOMException);
  // 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;
  // Introduced in DOM Level 3:
  readonly attribute TypeInfo        schemaTypeInfo;
  // Introduced in DOM Level 3:
  readonly attribute boolean         isId;
};

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)
                                      raises(DOMException);
  // 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)
                                      raises(DOMException);
  // 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)
                                      raises(DOMException);
  // Introduced in DOM Level 2:
  boolean            hasAttribute(in DOMString name);
  // Introduced in DOM Level 2:
  boolean            hasAttributeNS(in DOMString namespaceURI,
                                    in DOMString localName)
                                      raises(DOMException);
  // Introduced in DOM Level 3:
  readonly attribute TypeInfo        schemaTypeInfo;
  // Introduced in DOM Level 3:
  void               setIdAttribute(in DOMString name,
                                    in boolean isId)
                                      raises(DOMException);
  // Introduced in DOM Level 3:
  void               setIdAttributeNS(in DOMString namespaceURI,
                                      in DOMString localName,
                                      in boolean isId)
                                      raises(DOMException);
  // Introduced in DOM Level 3:
  void               setIdAttributeNode(in Attr idAttr,
                                        in boolean isId)
                                      raises(DOMException);
};
```

```
interface Text : CharacterData {
  Text               splitText(in unsigned long offset)
                                raises(DOMException);
  // Introduced in DOM Level 3:
  readonly attribute boolean         isElementContentWhitespace;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       wholeText;
  // Introduced in DOM Level 3:
  Text               replaceWholeText(in DOMString content)
                                raises(DOMException);
};

interface Comment : CharacterData {
};

// Introduced in DOM Level 3:
interface TypeInfo {
  readonly attribute DOMString       typeName;
  readonly attribute DOMString       typeNamespace;

  // DerivationMethods
  const unsigned long       DERIVATION_RESTRICTION         = 0x00000001;
  const unsigned long       DERIVATION_EXTENSION           = 0x00000002;
  const unsigned long       DERIVATION_UNION               = 0x00000004;
  const unsigned long       DERIVATION_LIST                = 0x00000008;

  boolean            isDerivedFrom(in DOMString typeNamespaceArg,
                                   in DOMString typeNameArg,
                                   in unsigned long derivationMethod);
};

// Introduced in DOM Level 3:
interface UserDataHandler {

  // OperationType
  const unsigned short      NODE_CLONED                    = 1;
  const unsigned short      NODE_IMPORTED                  = 2;
  const unsigned short      NODE_DELETED                   = 3;
  const unsigned short      NODE_RENAMED                   = 4;
  const unsigned short      NODE_ADOPTED                   = 5;

  void               handle(in unsigned short operation,
                            in DOMString key,
                            in DOMUserData data,
                            in Node src,
                            in Node dst);
};

// Introduced in DOM Level 3:
interface DOMError {

  // ErrorSeverity
  const unsigned short      SEVERITY_WARNING               = 1;
  const unsigned short      SEVERITY_ERROR                 = 2;
  const unsigned short      SEVERITY_FATAL_ERROR           = 3;
```

```
  readonly attribute unsigned short  severity;
  readonly attribute DOMString       message;
  readonly attribute DOMString       type;
  readonly attribute DOMObject       relatedException;
  readonly attribute DOMObject       relatedData;
  readonly attribute DOMLocator      location;
};

// Introduced in DOM Level 3:
interface DOMErrorHandler {
  boolean            handleError(in DOMError error);
};

// Introduced in DOM Level 3:
interface DOMLocator {
  readonly attribute long            lineNumber;
  readonly attribute long            columnNumber;
  readonly attribute long            byteOffset;
  readonly attribute long            utf16Offset;
  readonly attribute Node            relatedNode;
  readonly attribute DOMString       uri;
};

// Introduced in DOM Level 3:
interface DOMConfiguration {
  void               setParameter(in DOMString name,
                                  in DOMUserData value)
                                      raises(DOMException);
  DOMUserData        getParameter(in DOMString name)
                                      raises(DOMException);
  boolean            canSetParameter(in DOMString name,
                                     in DOMUserData value);
  readonly attribute DOMStringList   parameterNames;
};

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;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       inputEncoding;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       xmlEncoding;
  // Introduced in DOM Level 3:
  readonly attribute DOMString       xmlVersion;
};

interface EntityReference : Node {
};

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

};

interface DocumentFragment : Node {
};

interface Document : Node {
  // Modified in DOM Level 3:
  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);
    // Introduced in DOM Level 3:
    readonly attribute DOMString       inputEncoding;
    // Introduced in DOM Level 3:
    readonly attribute DOMString       xmlEncoding;
    // Introduced in DOM Level 3:
             attribute boolean         xmlStandalone;
                                        // raises(DOMException) on setting

    // Introduced in DOM Level 3:
             attribute DOMString       xmlVersion;
                                        // raises(DOMException) on setting

    // Introduced in DOM Level 3:
             attribute boolean         strictErrorChecking;
    // Introduced in DOM Level 3:
             attribute DOMString       documentURI;
    // Introduced in DOM Level 3:
    Node               adoptNode(in Node source)
                                         raises(DOMException);
    // Introduced in DOM Level 3:
    readonly attribute DOMConfiguration domConfig;
    // Introduced in DOM Level 3:
    void               normalizeDocument();
    // Introduced in DOM Level 3:
    Node               renameNode(in Node n,
                                  in DOMString namespaceURI,
                                  in DOMString qualifiedName)
                                        raises(DOMException);
  };
};

#endif // _DOM_IDL_
```

Appendix D: Configuration Settings

Editor:
Elena Litani, IBM

D.1 Configuration Scenarios

Using the `DOMConfiguration` [p.106] users can change behavior of the `DOMParser`, `DOMSerializer` and `Document.normalizeDocument()` [p.54] . If a DOM implementation supports XML Schemas and DTD validation, the table below defines behavior of such implementation following various parameter settings on the `DOMConfiguration`. Errors are effectively reported only if a `DOMErrorHandler` [p.105] object is attached to the "error-handler [p.108] " parameter.

"schema-type [p.110] "	"validate [p.110] "	"validate-if-schema [p.111] "	Instance schemas, i.e. the current schema	Outcome	Other parameters
`null`	`true`	`false`	DTD and XML Schema	Implementation dependent	The outcome of setting the "datatype-normalization [p.108] ", "element-content-whitespace [p.108] " or "namespaces [p.109] " parameters to `true` or `false` is implementation dependent.
	`false`	`true`			
`null`	`true`	`false`	none	Report an error	Setting the "datatype-normalization [p.108] " to `true` or `false` has no effect on the DOM.
	`false`	`true`		No error is reported	
`null`	`true`	`false`	DTD	Validate against DTD	Setting the "datatype-normalization [p.108] " to `true` or `false` has no effect on the DOM.
	`false`	`true`			
`null`	`true`	`false`	XML Schema	Validate against XML Schema	The outcome of setting the "namespaces [p.109] " to `false` is implementation dependent (likely to be an error). Setting the "element-content-whitespace [p.108] " to `false` does not have any effect on the DOM.
	`false`	`true`			
`"http://www.w3.org/TR/REC-xml"`	`true`	`false`	DTD or XML Schema or both	If DTD is found, validate against DTD. Otherwise, report an error.	Setting the "datatype-normalization [p.108] " to `true` or `false` has no effect on the DOM.
	`false`	`true`		If DTD is found, validate against DTD.	

`"http://www.w3.org/2001/XMLSchema"`	`true`	`false`	DTD or XML Schema or both	If XML Schema is found, validate against the schema. Otherwise, report an error.	Setting the "datatype-normalization [p.108] " to `true` exposes XML Schema normalized values in the DOM. The outcome of setting the "namespaces [p.109] " to `false` is implementation dependent (likely to be an error).
	`false`	`true`		If XML Schema is found, validate against the schema.	
`"http://www.w3.org/2001/XMLSchema"` or `"http://www.w3.org/TR/REC-xml"`	`false`	`false`	DTD or XML Schema or both	If XML Schema is found, it is ignored. DOM implementations may use information available in the DTD to perform entity resolution.	Setting the "datatype-normalization [p.108] " to `true` of `false` has no effect on the DOM.

Note: If an error has to be reported, as specified in the "Outcome" column above, the `DOMError.type` [p.105] is `"no-schema-available"`.

Appendix C: Infoset Mapping

Editor:
Philippe Le Hégaret, W3C

This appendix contains the mappings between the XML Information Set [*XML Information Set*] model and the Document Object Model. Starting from a `Document` [p.41] node, each *information item* is mapped to its respective `Node` [p.56] , and each `Node` is mapped to its respective *information item*. As used in the Infoset specification, the Infoset property names are shown in square brackets, **[thus]**.

Unless specified, the Infoset to DOM node mapping makes no distinction between unknown and no value since both will be exposed as `null` (or `false` if the DOM attribute is of type `boolean`).

C.1 Document Node Mapping

C.1.1 Infoset to Document Node

An *document information item* maps to a `Document` [p.41] node. The attributes of the corresponding `Document` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	`"#document"`
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.DOCUMENT_NODE` [p.58]
`Node.parentNode` [p.62]	`null`
`Node.childNodes` [p.61]	A `NodeList` [p.73] containing the information items in the **[children]** property.
`Node.firstChild` [p.61]	The first node contained in `Node.childNodes` [p.61]
`Node.lastChild` [p.61]	The last node contained in `Node.childNodes` [p.61]
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	`null`
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	same as `Document.documentURI` [p.42]

`Node.textContent` [p.63]	`null`
`Document.doctype` [p.42]	The document type information item
`Document.implementation` [p.43]	The `DOMImplementation` [p.37] object used to create this node
`Document.documentElement` [p.42]	The **[document element]** property
`Document.inputEncoding` [p.43]	The **[character encoding scheme]** property
`Document.xmlEncoding` [p.43]	`null`
`Document.xmlStandalone` [p.43]	The **[standalone]** property, or `false` if the latter has no value.
`Document.xmlVersion` [p.43]	The **[version]** property, or `"1.0"` if the latter has no value.
`Document.strictErrorChecking` [p.43]	`true`
`Document.documentURI` [p.42]	The **[base URI]** property
`Document.domConfig` [p.43]	A `DOMConfiguration` [p.106] object whose parameters are set to their default values

The **[notations]**, **[unparsed entities]** properties are being exposed in the `DocumentType` [p.115] node.

Note: The **[all declarations processed]** property is not exposed through the `Document` [p.41] node.

C.1.2 Document Node to Infoset

A `Document` [p.41] node maps to an *document information item*. `Document` nodes with no namespace URI (`Node.namespaceURI` [p.61] equals to `null`) cannot be represented using the Infoset. The properties of the corresponding *document information item* are constructed as follows:

Property	Value
[children]	`Node.childNodes` [p.61]
[document element]	`Document.documentElement` [p.42]
[notations]	`Document.doctype.notations`
[unparsed entities]	The information items from `Document.doctype.entities`, whose `Node.childNodes` [p.61] is an empty list
[base URI]	`Document.documentURI` [p.42]
[character encoding scheme]	`Document.inputEncoding` [p.43]
[standalone]	`Document.xmlStandalone` [p.43]
[version]	`Document.xmlVersion` [p.43]
[all declarations processed]	The value is implementation dependent

C.2 Element Node Mapping

C.2.1 Infoset to Element Node

An *element information item* maps to a `Element` [p.85] node. The attributes of the corresponding `Element` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	same as `Element.tagName` [p.86]
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.ELEMENT_NODE` [p.59]
`Node.parentNode` [p.62]	The **[parent]** property
`Node.childNodes` [p.61]	A `NodeList` [p.73] containing the information items in the **[children]** property
`Node.firstChild` [p.61]	The first node contained in `Node.childNodes` [p.61]
`Node.lastChild` [p.61]	The last node contained in `Node.childNodes` [p.61]
`Node.previousSibling` [p.63]	The information item preceding the current one on the **[children]** property contained in the **[parent]** property
`Node.nextSibling` [p.62]	The information item following the current one on the **[children]** property contained in the **[parent]** property
`Node.attributes` [p.61]	The information items contained in the **[attributes]** and **[namespace attributes]** properties
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	The **[namespace name]** property
`Node.prefix` [p.62]	The **[prefix]** property
`Node.localName` [p.61]	The **[local name]** property
`Node.baseURI` [p.61]	The **[base URI]** property
`Node.textContent` [p.63]	Concatenation of the `Node.textContent` [p.63] attribute value of every child node, excluding `COMMENT_NODE` and `PROCESSING_INSTRUCTION_NODE` nodes. This is the empty string if the node has no children.
`Element.tagName` [p.86]	If the **[prefix]** property has no value, this contains the **[local name]** property. Otherwise, this contains the concatenation of the **[prefix]** property, the colon ':' character, and the **[local name]** property.
`Element.schemaTypeInfo` [p.86]	A `TypeInfo` [p.99] object whose `TypeInfo.typeNamespace` [p.102] and `TypeInfo.typeName` [p.102] are inferred from the schema in use if available.

Note: The **[in-scope namespaces]** property is not exposed through the `Element` [p.85] node.

C.2.2 Element Node to Infoset

An `Element` [p.85] node maps to an *element information item*. Because the Infoset only represents unexpanded entity references, non-empty `EntityReference` [p.118] nodes contained in `Node.childNodes` [p.61] need to be replaced by their content. DOM applications could use the `Document.normalizeDocument()` [p.54] method for that effect with the "entities [p.108] " parameter set to `false`. The properties of the corresponding *element information item* are constructed as follows:

Property	Value
[namespace name]	`Node.namespaceURI` [p.61]
[local name]	`Node.localName` [p.61]
[prefix]	`Node.prefix` [p.62]
[children]	`Node.childNodes` [p.61] , whose expanded entity references (`EntityReference` [p.118] nodes with children) have been replaced with their content.
[attributes]	The nodes contained in `Node.attributes` [p.61] , whose `Node.namespaceURI` [p.61] value is different from `"http://www.w3.org/2000/xmlns/"`
[namespace attributes]	The nodes contained in `Node.attributes` [p.61] , whose `Node.namespaceURI` [p.61] value is `"http://www.w3.org/2000/xmlns/"`
[in-scope namespaces]	The namespace information items computed using the **[namespace attributes]** properties of this node and its ancestors. If the [*DOM Level 3 XPath*] module is supported, the namespace information items can also be computed from the `XPathNamespace` nodes.
[base URI]	`Node.baseURI` [p.61]
[parent]	`Node.parentNode` [p.62]

C.3 Attr Node Mapping

C.3.1 Infoset to Attr Node

An *attribute information item* map to a `Attr` [p.81] node. The attributes of the corresponding `Attr` node are constructed as follows:

Attribute/Method	Value
`Node.nodeName` [p.62]	same as `Attr.name` [p.84]
`Node.nodeValue` [p.62]	same as `Attr.value` [p.84]
`Node.nodeType` [p.62]	`Node.ATTRIBUTE_NODE` [p.58]
`Node.parentNode` [p.62]	`null`
`Node.childNodes` [p.61]	A `NodeList` [p.73] containing one `Text` [p.95] node whose text content is the same as `Attr.value` [p.84] .
`Node.firstChild` [p.61]	The `Text` [p.95] node contained in `Node.childNodes` [p.61]
`Node.lastChild` [p.61]	The `Text` [p.95] node contained in `Node.childNodes` [p.61]
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	The **[namespace name]** property
`Node.prefix` [p.62]	The **[prefix]** property
`Node.localName` [p.61]	The **[local name]** property
`Node.baseURI` [p.61]	`null`
`Node.textContent` [p.63]	the value of `Node.textContent` [p.63] of the `Text` [p.95] child. same as `Node.nodeValue` [p.62] (since this attribute node only contains one `Text` node)
`Attr.name` [p.84]	If the **[prefix]** property has no value, this contains the **[local name]** property. Otherwise, this contains the concatenation of the **[prefix]** property, the colon ':' character, and the **[local name]** property.
`Attr.specified` [p.84]	The **[specified]** property
`Attr.value` [p.84]	The **[normalized value]** property
`Attr.ownerElement` [p.84]	The **[owner element]** property
`Attr.schemaTypeInfo` [p.84]	A `TypeInfo` [p.99] object whose `TypeInfo.typeNamespace` [p.102] is `"http://www.w3.org/TR/REC-xml"` and `TypeInfo.typeName` [p.102] is the **[attribute type]** property
`Attr.isId` [p.83]	if the **[attribute type]** property is ID, this method return `true`

C.3.2 Attr Node to Infoset

An `Attr` [p.81] node maps to an *attribute information item*. `Attr` nodes with no namespace URI (`Node.namespaceURI` [p.61] equals to `null`) cannot be represented using the Infoset. The properties of the corresponding *attribute information item* are constructed as follows:

Property	Value
[namespace name]	`Node.namespaceURI` [p.61]
[local name]	`Node.localName` [p.61]
[prefix]	`Node.prefix` [p.62]
[normalized value]	`Attr.value` [p.84]
[specified]	`Attr.specified` [p.84]
[attribute type]	Using the `TypeInfo` [p.99] object referenced from `Attr.schemaTypeInfo` [p.84] , the value of `TypeInfo.typeName` [p.102] if `TypeInfo.typeNamespace` [p.102] is `"http://www.w3.org/TR/REC-xml"`.
[references]	if the computed **[attribute type]** property is IDREF, IDREFS, ENTITY, ENTITIES, or NOTATION, the value of this property is an ordered list of the element, unparsed entity, or notation information items referred to in the attribute value, in the order that they appear there. The ordered list is computed using `Node.ownerDocument.getElementById`, `Node.ownerDocument.doctype.entities`, and `Node.ownerDocument.doctype.notations`.
[owner element]	`Attr.ownerElement` [p.84]

C.4 ProcessingInstruction Node Mapping

C.4.1 Infoset to ProcessingInstruction Node

A *processing instruction information item* map to a `ProcessingInstruction` [p.118] node. The attributes of the corresponding `ProcessingInstruction` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	same as `ProcessingInstruction.target` [p.119]
`Node.nodeValue` [p.62]	same as `ProcessingInstruction.data` [p.119]
`Node.nodeType` [p.62]	`Node.PROCESSING_INSTRUCTION_NODE` [p.59]
`Node.parentNode` [p.62]	The **[parent]** property
`Node.childNodes` [p.61]	empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	The **[base URI]** property of the parent element if any. The **[base URI]** property of the processing instruction information item is not exposed through the `ProcessingInstruction` [p.118] node.
`Node.textContent` [p.63]	same as `Node.nodeValue` [p.62]
`ProcessingInstruction.target` [p.119]	The **[target]** property
`ProcessingInstruction.data` [p.119]	The **[content]** property

C.4.2 ProcessingInstruction Node to Infoset

A `ProcessingInstruction` [p.118] node maps to an *processing instruction information item*. The properties of the corresponding *processing instruction information item* are constructed as follows:

Property	Value
[target]	`ProcessingInstruction.target` [p.119]
[content]	`ProcessingInstruction.data` [p.119]
[base URI]	`Node.baseURI` [p.61] (which is equivalent to the base URI of its parent element if any)
[notation]	The `Notation` [p.116] node named by the target and if available from `Node.ownerDocument.doctype.notations`
[parent]	`Node.parentNode` [p.62]

C.5 EntityReference Node Mapping

C.5.1 Infoset to EntityReference Node

An *unexpanded entity reference information item* maps to a `EntityReference` [p.118] node. The attributes of the corresponding `EntityReference` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	The **[name]** property
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.ENTITY_REFERENCE_NODE` [p.59]
`Node.parentNode` [p.62]	the **[parent]** property
`Node.childNodes` [p.61]	Empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	The **[declaration base URI]** property
`Node.textContent` [p.63]	`null` (the node has no children)

Note: The **[system identifier]** and **[public identifier]** properties are not exposed through the `EntityReference` [p.118] node, but through the `Entity` [p.116] node reference from this `EntityReference` node, if any.

C.5.2 EntityReference Node to Infoset

An `EntityReference` [p.118] node maps to an *unexpanded entity reference information item*. `EntityReference` nodes with children (`Node.childNodes` [p.61] contains a non-empty list) cannot be represented using the Infoset. The properties of the corresponding *unexpanded entity reference information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName` [p.62]
[system identifier]	The `Entity.systemId` [p.117] value of the `Entity` [p.116] node available from `Node.ownerDocument.doctype.entities` if available
[public identifier]	The `Entity.publicId` [p.117] value of the `Entity` [p.116] node available from `Node.ownerDocument.doctype.entities` if available
[declaration base URI]	`Node.baseURI` [p.61]
[parent]	`Node.parentNode` [p.62]

C.6 Text and CDATASection Nodes Mapping

Since the [*XML Information Set*] doesn't represent the boundaries of CDATA marked sections, `CDATASection` [p.114] nodes cannot occur from an infoset mapping.

C.6.1 Infoset to Text Node

Consecutive *character information items* map to a `Text` [p.95] node. The attributes of the corresponding `Text` node are constructed as follows:

Attribute/Method	Value
`Node.nodeName` [p.62]	`"#text"`
`Node.nodeValue` [p.62]	same as `CharacterData.data` [p.79]
`Node.nodeType` [p.62]	`Node.TEXT_NODE` [p.59]
`Node.parentNode` [p.62]	The **[parent]** property
`Node.childNodes` [p.61]	empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	null
`Node.textContent` [p.63]	same as `Node.nodeValue` [p.62]
`CharacterData.data` [p.79]	A `DOMString` [p.24] including all **[character code]** contained in the *character information items*
`CharacterData.length` [p.79]	The number of 16-bit units needed to encode all ISO 10646 character code contained in the *character information items* using the UTF-16 encoding.
`Text.isElementContentWhitespace` [p.96]	The **[element content whitespace]** property
`Text.wholeText` [p.96]	same as `CharacterData.data` [p.79]

Note: By construction, the values of the **[parent]** and **[element content whitespace]** properties are necessarily the sames for all consecutive *character information items*.

C.6.2 Text and CDATASection Nodes to Infoset

The text content of a `Text` [p.95] or a `CDATASection` [p.114] node maps to a sequence of *character information items*. The number of items is less or equal to `CharacterData.length` [p.79] . Text nodes contained in `Attr` [p.81] nodes are mapped to the Infoset using the `Attr.value` [p.84] attribute. Text nodes contained in `Document` [p.41] nodes cannot be represented using the Infoset. The properties of the corresponding *character information items* are constructed as follows:

Property	Value
[character code]	The ISO 10646 character code produced using one or two *16-bit units* from `CharacterData.data` [p.79]
[element content whitespace]	`Text.isElementContentWhitespace` [p.96]
[parent]	`Node.parentNode` [p.62]

C.7 Comment Node Mapping

C.7.1 Infoset to Comment Node

A *comment information item* maps to a `Comment` [p.99] node. The attributes of the corresponding `Comment` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	`"#comment"`
`Node.nodeValue` [p.62]	same as `CharacterData.data` [p.79]
`Node.nodeType` [p.62]	`Node.COMMENT_NODE` [p.58]
`Node.parentNode` [p.62]	The **[parent]** property
`Node.childNodes` [p.61]	empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	`null`
`Node.textContent` [p.63]	same as `Node.nodeValue` [p.62]
`CharacterData.data` [p.79]	The **[content]** property encoded using the UTF-16 encoding.
`CharacterData.length` [p.79]	The number of 16-bit units needed to encode all ISO character code contained in the **[content]** property using the UTF-16 encoding.

C.7.2 Comment Node to Infoset

A `Comment` [p.99] maps to a *comment information item*. The properties of the corresponding *comment information item* are constructed as follows:

Property	Value
[content]	`CharacterData.data` [p.79]
[parent]	`Node.parentNode` [p.62]

C.8 DocumentType Node Mapping

C.8.1 Infoset to DocumentType Node

A *document type declaration information item* maps to a `DocumentType` [p.115] node. The attributes of the corresponding `DocumentType` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	same as `DocumentType.name` [p.116]
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.DOCUMENT_TYPE_NODE` [p.59]
`Node.parentNode` [p.62]	The **[parent]** property
`Node.childNodes` [p.61]	empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	`null`
`Node.textContent` [p.63]	`null`
`DocumentType.name` [p.116]	The name of the document element.
`DocumentType.entities` [p.115]	The **[unparsed entities]** property available from the document information item.
`DocumentType.notations` [p.116]	The **[notations]** property available from the document information item.
`DocumentType.publicId` [p.116]	The **[public identifier]** property
`DocumentType.systemId` [p.116]	The **[system identifier]** property
`DocumentType.internalSubset` [p.115]	The value is implementation dependent

Note: The **[children]** property is not exposed through the `DocumentType` [p.115] node.

C.8.2 DocumentType Node to Infoset

A `DocumentType` [p.115] maps to a *document type declaration information item*. The properties of the corresponding *document type declaration information item* are constructed as follows:

Property	Value
[system identifier]	`DocumentType.systemId` [p.116]
[public identifier]	`DocumentType.publicId` [p.116]
[children]	The value of this property is implementation dependent
[parent]	`Node.parentNode` [p.62]

C.9 Entity Node Mapping

C.9.1 Infoset to Entity Node

An *unparsed entity information item* maps to a `Entity` [p.116] node. The attributes of the corresponding `Entity` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	The **[name]** property
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.ENTITY_NODE` [p.59]
`Node.parentNode` [p.62]	`null`
`Node.childNodes` [p.61]	Empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	The **[declaration base URI]** property
`Node.textContent` [p.63]	`""` (the node has no children)
`Entity.publicId` [p.117]	The **[public identifier]** property
`Entity.systemId` [p.117]	The **[system identifier]** property
`Entity.notationName` [p.117]	The **[notation name]** property
`Entity.inputEncoding` [p.117]	`null`
`Entity.xmlEncoding` [p.118]	`null`
`Entity.xmlVersion` [p.118]	`null`

Note: The **[notation]** property is available through the `DocumentType` [p.115] node.

C.9.2 Entity Node to Infoset

An `Entity` [p.116] node maps to an *unparsed entity information item*. `Entity` nodes with children (`Node.childNodes` [p.61] contains a non-empty list) cannot be represented using the Infoset. The properties of the corresponding *unparsed entity information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName` [p.62]
[system identifier]	`Entity.systemId` [p.117]
[public identifier]	`Entity.publicId` [p.117]
[declaration base URI]	`Node.baseURI` [p.61]
[notation name]	`Entity.notationName` [p.117]
[notation]	The `Notation` [p.116] node referenced from `DocumentType.notations` [p.116] whose name is the **[notation name]** property

C.10 Notation Node Mapping

C.10.1 Infoset to Notation Node

A *notation information item* maps to a `Notation` [p.116] node. The attributes of the corresponding `Notation` node are constructed as follows:

Attribute	Value
`Node.nodeName` [p.62]	The **[name]** property
`Node.nodeValue` [p.62]	`null`
`Node.nodeType` [p.62]	`Node.NOTATION_NODE` [p.59]
`Node.parentNode` [p.62]	`null`
`Node.childNodes` [p.61]	Empty `NodeList` [p.73]
`Node.firstChild` [p.61]	`null`
`Node.lastChild` [p.61]	`null`
`Node.previousSibling` [p.63]	`null`
`Node.nextSibling` [p.62]	`null`
`Node.attributes` [p.61]	`null`
`Node.ownerDocument` [p.62]	The document information item
`Node.namespaceURI` [p.61]	`null`
`Node.prefix` [p.62]	`null`
`Node.localName` [p.61]	`null`
`Node.baseURI` [p.61]	The **[declaration base URI]** property
`Node.textContent` [p.63]	`null`
`Notation.publicId` [p.116]	The **[public identifier]** property
`Notation.systemId` [p.116]	The **[system identifier]** property

C.10.2 Notation Node to Infoset

A `Notation` [p.116] maps to a *notation information item*. The properties of the corresponding *notation information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName` [p.62]
[system identifier]	`Notation.systemId` [p.116]
[public identifier]	`Notation.publicId` [p.116]
[parent]	`Node.parentNode` [p.62]

Appendix G: Java Language Binding

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

The Java files are also available as
http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/java-binding.zip

G.1 Java Binding Extension

Note: This section is informative.

This section defines the `DOMImplementationRegistry` object, discussed in Bootstrapping [p.30] , for Java.

The `DOMImplementationRegistry` is first initialized by the application or the implementation, depending on the context, through the Java system property "org.w3c.dom.DOMImplementationSourceList". The value of this property is a space separated list of names of available classes implementing the `DOMImplementationSource` [p.36] interface.

org/w3c/dom/bootstrap/DOMImplementationRegistry.java:

```
package org.w3c.dom.bootstrap;

import java.util.StringTokenizer;
import java.util.Vector;
import org.w3c.dom.DOMImplementationSource;
import org.w3c.dom.DOMImplementationList;
import org.w3c.dom.DOMImplementation;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.security.AccessController;
import java.security.PrivilegedAction;

/**
 * A factory that enables applications to obtain instances of
 * <code>DOMImplementation</code>.
 *
 * <p>
 * Example:
 * </p>
 *
 * <pre class='example'>
 *  // get an instance of the DOMImplementation registry
 *  DOMImplementationRegistry registry =
 *       DOMImplementationRegistry.newInstance();
 *  // get a DOM implementation the Level 3 XML module
 *  DOMImplementation domImpl =
 *       registry.getDOMImplementation("XML 3.0");
 * </pre>
 *
 * <p>
```

```
 * This provides an application with an implementation-independent starting
 * point. DOM implementations may modify this class to meet new security
 * standards or to provide *additional* fallbacks for the list of
 * DOMImplementationSources.
 * </p>
 *
 * @see DOMImplementation
 * @see DOMImplementationSource
 * @since DOM Level 3
 */
public final class DOMImplementationRegistry {
    /**
     * The system property to specify the
     * DOMImplementationSource class names.
     */
    public static final String PROPERTY =
        "org.w3c.dom.DOMImplementationSourceList";

    /**
     * Default columns per line.
     */
    private static final int DEFAULT_LINE_LENGTH = 80;

    /**
     * The list of DOMImplementationSources.
     */
    private Vector sources;

    /**
     * Private constructor.
     * @param srcs Vector List of DOMImplementationSources
     */
    private DOMImplementationRegistry(final Vector srcs) {
        sources = srcs;
    }

    /**
     * Obtain a new instance of a <code>DOMImplementationRegistry</code>.
     *

     * The <code>DOMImplementationRegistry</code> is initialized by the
     * application or the implementation, depending on the context, by
     * first checking the value of the Java system property
     * <code>org.w3c.dom.DOMImplementationSourceList</code> and
     * the the service provider whose contents are at
     * "<code>META_INF/services/org.w3c.dom.DOMImplementationSourceList</code>"
     * The value of this property is a white-space separated list of
     * names of availables classes implementing the
     * <code>DOMImplementationSource</code> interface. Each class listed
     * in the class name list is instantiated and any exceptions
     * encountered are thrown to the application.
     *
     * @return an initialized instance of DOMImplementationRegistry
     * @throws ClassNotFoundException
     *     If any specified class can not be found
     * @throws InstantiationException
     *     If any specified class is an interface or abstract class
```

```
     * @throws IllegalAccessException
     *     If the default constructor of a specified class is not accessible
     * @throws ClassCastException
     *     If any specified class does not implement
     * <code>DOMImplementationSource</code>
     */
    public static DOMImplementationRegistry newInstance()
        throws
        ClassNotFoundException,
        InstantiationException,
        IllegalAccessException,
        ClassCastException {
        Vector sources = new Vector();

        ClassLoader classLoader = getClassLoader();
        // fetch system property:
        String p = getSystemProperty(PROPERTY);

        //
        // if property is not specified then use contents of
        // META_INF/org.w3c.dom.DOMImplementationSourceList from classpath
        if (p == null) {
            p = getServiceValue(classLoader);
        }
        if (p == null) {
            //
            // DOM Implementations can modify here to add *additional* fallback
            // mechanisms to access a list of default DOMImplementationSources.

        }
        if (p != null) {
            StringTokenizer st = new StringTokenizer(p);
            while (st.hasMoreTokens()) {
                String sourceName = st.nextToken();
                // Use context class loader, falling back to Class.forName
                // if and only if this fails...
                Class sourceClass = null;
                if (classLoader != null) {
                    sourceClass = classLoader.loadClass(sourceName);
                } else {
                    sourceClass = Class.forName(sourceName);
                }
                DOMImplementationSource source =
                    (DOMImplementationSource) sourceClass.newInstance();
                sources.addElement(source);
            }
        }
        return new DOMImplementationRegistry(sources);
    }

    /**
     * Return the first implementation that has the desired
     * features, or <code>null</code> if none is found.
     *
     * @param features
     *            A string that specifies which features are required. This is
     *            a space separated list in which each feature is specified by
```

```
     *            its name optionally followed by a space and a version number.
     *            This is something like: "XML 1.0 Traversal +Events 2.0"
     * @return An implementation that has the desired features,
     *         or <code>null</code> if none found.
     */
    public DOMImplementation getDOMImplementation(final String features) {
        int size = sources.size();
        String name = null;
        for (int i = 0; i < size; i++) {
            DOMImplementationSource source =
                (DOMImplementationSource) sources.elementAt(i);
            DOMImplementation impl = source.getDOMImplementation(features);
            if (impl != null) {
                return impl;
            }
        }
        return null;
    }

    /**
     * Return a list of implementations that support the
     * desired features.
     *
     * @param features
     *            A string that specifies which features are required. This is
     *            a space separated list in which each feature is specified by
     *            its name optionally followed by a space and a version number.
     *            This is something like: "XML 1.0 Traversal +Events 2.0"
     * @return A list of DOMImplementations that support the desired features.
     */
    public DOMImplementationList getDOMImplementationList(final String features) {
        final Vector implementations = new Vector();
        int size = sources.size();
        for (int i = 0; i < size; i++) {
            DOMImplementationSource source =
                (DOMImplementationSource) sources.elementAt(i);
            DOMImplementationList impls =
                source.getDOMImplementationList(features);
            for (int j = 0; j < impls.getLength(); j++) {
                DOMImplementation impl = impls.item(j);
                implementations.addElement(impl);
            }
        }
        return new DOMImplementationList() {
                public DOMImplementation item(final int index) {
                    if (index >= 0 && index < implementations.size()) {
                        try {
                            return (DOMImplementation)
                                implementations.elementAt(index);
                        } catch (ArrayIndexOutOfBoundsException e) {
                            return null;
                        }
                    }
                    return null;
                }

                public int getLength() {
```

```
                return implementations.size();
            }
        };
}

/**
 * Register an implementation.
 *
 * @param s The source to be registered, may not be <code>null</code>
 */
public void addSource(final DOMImplementationSource s) {
    if (s == null) {
        throw new NullPointerException();
    }
    if (!sources.contains(s)) {
        sources.addElement(s);
    }
}

/**
 *
 * Gets a class loader.
 *
 * @return A class loader, possibly <code>null</code>
 */
private static ClassLoader getClassLoader() {
    try {
        ClassLoader contextClassLoader = getContextClassLoader();

        if (contextClassLoader != null) {
            return contextClassLoader;
        }
    } catch (Exception e) {
        // Assume that the DOM application is in a JRE 1.1, use the
        // current ClassLoader
        return DOMImplementationRegistry.class.getClassLoader();
    }
    return DOMImplementationRegistry.class.getClassLoader();
}

/**
 * This method attempts to return the first line of the resource
 * META_INF/services/org.w3c.dom.DOMImplementationSourceList
 * from the provided ClassLoader.
 *
 * @param classLoader classLoader, may not be <code>null</code>.
 * @return first line of resource, or <code>null</code>
 */
private static String getServiceValue(final ClassLoader classLoader) {
    String serviceId = "META-INF/services/" + PROPERTY;
    // try to find services in CLASSPATH
    try {
        InputStream is = getResourceAsStream(classLoader, serviceId);

        if (is != null) {
            BufferedReader rd;
            try {
```

```
                rd =
                    new BufferedReader(new InputStreamReader(is, "UTF-8"),
                                       DEFAULT_LINE_LENGTH);
            } catch (java.io.UnsupportedEncodingException e) {
                rd =
                    new BufferedReader(new InputStreamReader(is),
                                       DEFAULT_LINE_LENGTH);
            }
            String serviceValue = rd.readLine();
            rd.close();
            if (serviceValue != null && serviceValue.length() > 0) {
                return serviceValue;
            }
        }
    } catch (Exception ex) {
        return null;
    }
    return null;
}

/**
 * A simple JRE (Java Runtime Environment) 1.1 test
 *
 * @return <code>true</code> if JRE 1.1
 */
private static boolean isJRE11() {
    try {
        Class c = Class.forName("java.security.AccessController");
        // java.security.AccessController existed since 1.2 so, if no
        // exception was thrown, the DOM application is running in a JRE
        // 1.2 or higher
        return false;
    } catch (Exception ex) {
        // ignore
    }
    return true;
}

/**
 * This method returns the ContextClassLoader or <code>null</code> if
 * running in a JRE 1.1
 *
 * @return The Context Classloader
 */
private static ClassLoader getContextClassLoader() {
    return isJRE11()
        ? null
        : (ClassLoader)
          AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    ClassLoader classLoader = null;
                    try {
                        classLoader =
                            Thread.currentThread().getContextClassLoader();
                    } catch (SecurityException ex) {
                    }
                    return classLoader;
```

```
                }
            });
}

/**
 * This method returns the system property indicated by the specified name
 * after checking access control privileges. For a JRE 1.1, this check is
 * not done.
 *
 * @param name the name of the system property
 * @return the system property
 */
private static String getSystemProperty(final String name) {
    return isJRE11()
        ? (String) System.getProperty(name)
        : (String) AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    return System.getProperty(name);
                }
            });
}

/**
 * This method returns an Inputstream for the reading resource
 * META_INF/services/org.w3c.dom.DOMImplementationSourceList after checking
 * access control privileges. For a JRE 1.1, this check is not done.
 *
 * @param classLoader classLoader
 * @param name the resource
 * @return an Inputstream for the resource specified
 */
private static InputStream getResourceAsStream(final ClassLoader classLoader,
                                               final String name) {
    if (isJRE11()) {
        InputStream ris;
        if (classLoader == null) {
            ris = ClassLoader.getSystemResourceAsStream(name);
        } else {
            ris = classLoader.getResourceAsStream(name);
        }
        return ris;
    } else {
        return (InputStream)
            AccessController.doPrivileged(new PrivilegedAction() {
                    public Object run() {
                        InputStream ris;
                        if (classLoader == null) {
                            ris =
                                ClassLoader.getSystemResourceAsStream(name);
                        } else {
                            ris = classLoader.getResourceAsStream(name);
                        }
                        return ris;
                    }
```

```
                    });
        }
    }
}
```

G.2 Other Core interfaces

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;
    public static final short VALIDATION_ERR            = 16;
    public static final short TYPE_MISMATCH_ERR         = 17;

}
```

org/w3c/dom/DOMStringList.java:

```
package org.w3c.dom;

public interface DOMStringList {
    public String item(int index);

    public int getLength();

    public boolean contains(String str);

}
```

org/w3c/dom/NameList.java:

```
package org.w3c.dom;

public interface NameList {
    public String getName(int index);

    public String getNamespaceURI(int index);

    public int getLength();

    public boolean contains(String str);

    public boolean containsNS(String namespaceURI,
                              String name);

}
```

org/w3c/dom/DOMImplementationList.java:

```
package org.w3c.dom;

public interface DOMImplementationList {
    public DOMImplementation item(int index);

    public int getLength();

}
```

org/w3c/dom/DOMImplementationSource.java:

```
package org.w3c.dom;

public interface DOMImplementationSource {
    public DOMImplementation getDOMImplementation(String features);

    public DOMImplementationList getDOMImplementationList(String features);

}
```

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;

    public Object getFeature(String feature,
                             String version);

}
```

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);

    public String getInputEncoding();

    public String getXmlEncoding();

    public boolean getXmlStandalone();
    public void setXmlStandalone(boolean xmlStandalone)
                                  throws DOMException;

    public String getXmlVersion();
    public void setXmlVersion(String xmlVersion)
                                  throws DOMException;

    public boolean getStrictErrorChecking();
    public void setStrictErrorChecking(boolean strictErrorChecking);

    public String getDocumentURI();
    public void setDocumentURI(String documentURI);

    public Node adoptNode(Node source)
                         throws DOMException;

    public DOMConfiguration getDomConfig();

    public void normalizeDocument();

    public Node renameNode(Node n,
                           String namespaceURI,
                           String qualifiedName)
                           throws DOMException;

}
```

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();

    public String getBaseURI();

    // DocumentPosition
    public static final short DOCUMENT_POSITION_DISCONNECTED = 0x01;
    public static final short DOCUMENT_POSITION_PRECEDING = 0x02;
    public static final short DOCUMENT_POSITION_FOLLOWING = 0x04;
    public static final short DOCUMENT_POSITION_CONTAINS = 0x08;
    public static final short DOCUMENT_POSITION_CONTAINED_BY = 0x10;
    public static final short DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;

    public short compareDocumentPosition(Node other)
                                         throws DOMException;

    public String getTextContent()
                                         throws DOMException;
    public void setTextContent(String textContent)
                                         throws DOMException;

    public boolean isSameNode(Node other);

    public String lookupPrefix(String namespaceURI);

    public boolean isDefaultNamespace(String namespaceURI);

    public String lookupNamespaceURI(String prefix);

    public boolean isEqualNode(Node arg);

    public Object getFeature(String feature,
                             String version);

    public Object setUserData(String key,
                              Object data,
                              UserDataHandler handler);

    public Object getUserData(String key);

}
```

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)
                               throws DOMException;

    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();

    public TypeInfo getSchemaTypeInfo();

    public boolean isId();

}
```

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)
                                throws DOMException;

    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)
                                   throws DOMException;

    public Attr setAttributeNodeNS(Attr newAttr)
                                   throws DOMException;

    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName)
                                           throws DOMException;

    public boolean hasAttribute(String name);

    public boolean hasAttributeNS(String namespaceURI,
                                  String localName)
                                  throws DOMException;

    public TypeInfo getSchemaTypeInfo();

    public void setIdAttribute(String name,
                               boolean isId)
                               throws DOMException;

    public void setIdAttributeNS(String namespaceURI,
                                 String localName,
                                 boolean isId)
                                 throws DOMException;

    public void setIdAttributeNode(Attr idAttr,
                                   boolean isId)
                                   throws DOMException;

}
```

org/w3c/dom/Text.java:

```
package org.w3c.dom;

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

    public boolean isElementContentWhitespace();
```

```
    public String getWholeText();

    public Text replaceWholeText(String content)
                                 throws DOMException;

}
```

org/w3c/dom/Comment.java:

```
package org.w3c.dom;

public interface Comment extends CharacterData {
}
```

org/w3c/dom/TypeInfo.java:

```
package org.w3c.dom;

public interface TypeInfo {
    public String getTypeName();

    public String getTypeNamespace();

    // DerivationMethods
    public static final int DERIVATION_RESTRICTION    = 0x00000001;
    public static final int DERIVATION_EXTENSION      = 0x00000002;
    public static final int DERIVATION_UNION          = 0x00000004;
    public static final int DERIVATION_LIST           = 0x00000008;

    public boolean isDerivedFrom(String typeNamespaceArg,
                                 String typeNameArg,
                                 int derivationMethod);

}
```

org/w3c/dom/UserDataHandler.java:

```
package org.w3c.dom;

public interface UserDataHandler {
    // OperationType
    public static final short NODE_CLONED              = 1;
    public static final short NODE_IMPORTED            = 2;
    public static final short NODE_DELETED             = 3;
    public static final short NODE_RENAMED             = 4;
    public static final short NODE_ADOPTED             = 5;

    public void handle(short operation,
                       String key,
                       Object data,
                       Node src,
                       Node dst);

}
```

org/w3c/dom/DOMError.java:

```
package org.w3c.dom;

public interface DOMError {
    // ErrorSeverity
    public static final short SEVERITY_WARNING          = 1;
    public static final short SEVERITY_ERROR            = 2;
    public static final short SEVERITY_FATAL_ERROR      = 3;

    public short getSeverity();

    public String getMessage();

    public String getType();

    public Object getRelatedException();

    public Object getRelatedData();

    public DOMLocator getLocation();

}
```

org/w3c/dom/DOMErrorHandler.java:

```
package org.w3c.dom;

public interface DOMErrorHandler {
    public boolean handleError(DOMError error);

}
```

org/w3c/dom/DOMLocator.java:

```
package org.w3c.dom;

public interface DOMLocator {
    public int getLineNumber();

    public int getColumnNumber();

    public int getByteOffset();

    public int getUtf16Offset();

    public Node getRelatedNode();

    public String getUri();

}
```

org/w3c/dom/DOMConfiguration.java:

```
package org.w3c.dom;

public interface DOMConfiguration {
    public void setParameter(String name,
                             Object value)
                             throws DOMException;

    public Object getParameter(String name)
                               throws DOMException;

    public boolean canSetParameter(String name,
                                   Object value);

    public DOMStringList getParameterNames();

}
```

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();

    public String getInputEncoding();

    public String getXmlEncoding();

    public String getXmlVersion();

}
```

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 H: ECMAScript Language Binding

This appendix contains the complete ECMAScript [*ECMAScript*] binding for the Level 3 Document Object Model Core definitions.

H.1 ECMAScript Binding Extension

This section defines the `DOMImplementationRegistry` object, discussed in Bootstrapping [p.30] , for ECMAScript.

Objects that implements the DOMImplementationRegistry interface

DOMImplementationRegistry is a global variable which has the following functions:

getDOMImplementation(features)

This method returns the first registered object that implements the **DOMImplementation** interface and has the desired features, or **null** if none is found.

The **features** parameter is a **String**. See also `DOMImplementationSource.getDOMImplementation()` [p.36] .

getDOMImplementationList(features)

This method returns a `DOMImplementationList` [p.35] list of registered object that implements the **DOMImplementation** interface and has the desired features.

The **features** parameter is a **String**. See also `DOMImplementationSource.getDOMImplementationList()` [p.37] .

H.2 Other Core interfaces

Properties of the **DOMException** Constructor function:

DOMException.INDEX_SIZE_ERR

The value of the constant **DOMException.INDEX_SIZE_ERR** is **1**.

DOMException.DOMSTRING_SIZE_ERR

The value of the constant **DOMException.DOMSTRING_SIZE_ERR** is **2**.

DOMException.HIERARCHY_REQUEST_ERR

The value of the constant **DOMException.HIERARCHY_REQUEST_ERR** is **3**.

DOMException.WRONG_DOCUMENT_ERR

The value of the constant **DOMException.WRONG_DOCUMENT_ERR** is **4**.

DOMException.INVALID_CHARACTER_ERR

The value of the constant **DOMException.INVALID_CHARACTER_ERR** is **5**.

DOMException.NO_DATA_ALLOWED_ERR

The value of the constant **DOMException.NO_DATA_ALLOWED_ERR** is **6**.

DOMException.NO_MODIFICATION_ALLOWED_ERR

The value of the constant **DOMException.NO_MODIFICATION_ALLOWED_ERR** is **7**.

DOMException.NOT_FOUND_ERR

The value of the constant **DOMException.NOT_FOUND_ERR** is **8**.

DOMException.NOT_SUPPORTED_ERR

The value of the constant **DOMException.NOT_SUPPORTED_ERR** is **9**.

DOMException.INUSE_ATTRIBUTE_ERR
The value of the constant **DOMException.INUSE_ATTRIBUTE_ERR** is **10**.
DOMException.INVALID_STATE_ERR
The value of the constant **DOMException.INVALID_STATE_ERR** is **11**.
DOMException.SYNTAX_ERR
The value of the constant **DOMException.SYNTAX_ERR** is **12**.
DOMException.INVALID_MODIFICATION_ERR
The value of the constant **DOMException.INVALID_MODIFICATION_ERR** is **13**.
DOMException.NAMESPACE_ERR
The value of the constant **DOMException.NAMESPACE_ERR** is **14**.
DOMException.INVALID_ACCESS_ERR
The value of the constant **DOMException.INVALID_ACCESS_ERR** is **15**.
DOMException.VALIDATION_ERR
The value of the constant **DOMException.VALIDATION_ERR** is **16**.
DOMException.TYPE_MISMATCH_ERR
The value of the constant **DOMException.TYPE_MISMATCH_ERR** is **17**.

Objects that implement the **DOMException** interface:

Properties of objects that implement the **DOMException** interface:

code
This property is a **Number**.

Objects that implement the **DOMStringList** interface:

Properties of objects that implement the **DOMStringList** interface:

length
This read-only property is a **Number**.

Functions of objects that implement the **DOMStringList** interface:

item(index)
This function returns a **String**.
The **index** parameter is a **Number**.
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** function with that index.

contains(str)
This function returns a **Boolean**.
The **str** parameter is a **String**.

Objects that implement the **NameList** interface:

Properties of objects that implement the **NameList** interface:

length
This read-only property is a **Number**.

Functions of objects that implement the **NameList** interface:

getName(index)
This function returns a **String**.
The **index** parameter is a **Number**.

getNamespaceURI(index)
This function returns a **String**.
The **index** parameter is a **Number**.

contains(str)
This function returns a **Boolean**.
The **str** parameter is a **String**.
containsNS(namespaceURI, name)
This function returns a **Boolean**.
The **namespaceURI** parameter is a **String**.
The **name** parameter is a **String**.

Objects that implement the **DOMImplementationList** interface:

Properties of objects that implement the **DOMImplementationList** interface:

length
This read-only property is a **Number**.

Functions of objects that implement the **DOMImplementationList** interface:

item(index)
This function returns an object that implements the **DOMImplementation** interface.
The **index** parameter is a **Number**.
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** function with that index.

Objects that implement the **DOMImplementationSource** interface:

Functions of objects that implement the **DOMImplementationSource** interface:

getDOMImplementation(features)
This function returns an object that implements the **DOMImplementation** interface.
The **features** parameter is a **String**.
getDOMImplementationList(features)
This function returns an object that implements the **DOMImplementationList** interface.
The **features** parameter is a **String**.

Objects that implement the **DOMImplementation** interface:

Functions of objects that implement the **DOMImplementation** interface:

hasFeature(feature, version)
This function returns a **Boolean**.
The **feature** parameter is a **String**.
The **version** parameter is a **String**.
createDocumentType(qualifiedName, publicId, systemId)
This function returns an object that implements the **DocumentType** interface.
The **qualifiedName** parameter is a **String**.
The **publicId** parameter is a **String**.
The **systemId** parameter is a **String**.
This function can raise an object that implements the **DOMException** interface.
createDocument(namespaceURI, qualifiedName, doctype)
This function returns an object that implements the **Document** interface.
The **namespaceURI** parameter is a **String**.
The **qualifiedName** parameter is a **String**.
The **doctype** parameter is an object that implements the **DocumentType** interface.
This function can raise an object that implements the **DOMException** interface.
getFeature(feature, version)
This function returns an object that implements the **Object** interface.

The **feature** parameter is a **String**.
The **version** parameter is a **String**.

Objects that implement the **DocumentFragment** interface:

Objects that implement the **DocumentFragment** interface have all properties and functions of the **Node** interface.

Objects that implement the **Document** interface:

Objects that implement the **Document** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **Document** interface:

doctype
This read-only property is an object that implements the **DocumentType** interface.

implementation
This read-only property is an object that implements the **DOMImplementation** interface.

documentElement
This read-only property is an object that implements the **Element** interface.

inputEncoding
This read-only property is a **String**.

xmlEncoding
This read-only property is a **String**.

xmlStandalone
This property is a **Boolean** and can raise an object that implements the **DOMException** interface on setting.

xmlVersion
This property is a **String** and can raise an object that implements the **DOMException** interface on setting.

strictErrorChecking
This property is a **Boolean**.

documentURI
This property is a **String**.

domConfig
This read-only property is an object that implements the **DOMConfiguration** interface.

Functions of objects that implement the **Document** interface:

createElement(tagName)
This function returns an object that implements the **Element** interface.
The **tagName** parameter is a **String**.
This function can raise an object that implements the **DOMException** interface.

createDocumentFragment()
This function returns an object that implements the **DocumentFragment** interface.

createTextNode(data)
This function returns an object that implements the **Text** interface.
The **data** parameter is a **String**.

createComment(data)
This function returns an object that implements the **Comment** interface.
The **data** parameter is a **String**.

createCDATASection(data)
This function returns an object that implements the **CDATASection** interface.

The **data** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

createProcessingInstruction(target, data)

This function returns an object that implements the **ProcessingInstruction** interface.

The **target** parameter is a **String**.

The **data** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

createAttribute(name)

This function returns an object that implements the **Attr** interface.

The **name** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

createEntityReference(name)

This function returns an object that implements the **EntityReference** interface.

The **name** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

getElementsByTagName(tagname)

This function returns an object that implements the **NodeList** interface.

The **tagname** parameter is a **String**.

importNode(importedNode, deep)

This function returns an object that implements the **Node** interface.

The **importedNode** parameter is an object that implements the **Node** interface.

The **deep** parameter is a **Boolean**.

This function can raise an object that implements the **DOMException** interface.

createElementNS(namespaceURI, qualifiedName)

This function returns an object that implements the **Element** interface.

The **namespaceURI** parameter is a **String**.

The **qualifiedName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

createAttributeNS(namespaceURI, qualifiedName)

This function returns an object that implements the **Attr** interface.

The **namespaceURI** parameter is a **String**.

The **qualifiedName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

getElementsByTagNameNS(namespaceURI, localName)

This function returns an object that implements the **NodeList** interface.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

getElementById(elementId)

This function returns an object that implements the **Element** interface.

The **elementId** parameter is a **String**.

adoptNode(source)

This function returns an object that implements the **Node** interface.

The **source** parameter is an object that implements the **Node** interface.

This function can raise an object that implements the **DOMException** interface.

normalizeDocument()

This function has no return value.

renameNode(n, namespaceURI, qualifiedName)

This function returns an object that implements the **Node** interface.

The **n** parameter is an object that implements the **Node** interface.

The **namespaceURI** parameter is a **String**.

The **qualifiedName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

Properties of the **Node** Constructor function:

Node.ELEMENT_NODE

The value of the constant **Node.ELEMENT_NODE** is **1**.

Node.ATTRIBUTE_NODE

The value of the constant **Node.ATTRIBUTE_NODE** is **2**.

Node.TEXT_NODE

The value of the constant **Node.TEXT_NODE** is **3**.

Node.CDATA_SECTION_NODE

The value of the constant **Node.CDATA_SECTION_NODE** is **4**.

Node.ENTITY_REFERENCE_NODE

The value of the constant **Node.ENTITY_REFERENCE_NODE** is **5**.

Node.ENTITY_NODE

The value of the constant **Node.ENTITY_NODE** is **6**.

Node.PROCESSING_INSTRUCTION_NODE

The value of the constant **Node.PROCESSING_INSTRUCTION_NODE** is **7**.

Node.COMMENT_NODE

The value of the constant **Node.COMMENT_NODE** is **8**.

Node.DOCUMENT_NODE

The value of the constant **Node.DOCUMENT_NODE** is **9**.

Node.DOCUMENT_TYPE_NODE

The value of the constant **Node.DOCUMENT_TYPE_NODE** is **10**.

Node.DOCUMENT_FRAGMENT_NODE

The value of the constant **Node.DOCUMENT_FRAGMENT_NODE** is **11**.

Node.NOTATION_NODE

The value of the constant **Node.NOTATION_NODE** is **12**.

Node.DOCUMENT_POSITION_DISCONNECTED

The value of the constant **Node.DOCUMENT_POSITION_DISCONNECTED** is **0x01**.

Node.DOCUMENT_POSITION_PRECEDING

The value of the constant **Node.DOCUMENT_POSITION_PRECEDING** is **0x02**.

Node.DOCUMENT_POSITION_FOLLOWING

The value of the constant **Node.DOCUMENT_POSITION_FOLLOWING** is **0x04**.

Node.DOCUMENT_POSITION_CONTAINS

The value of the constant **Node.DOCUMENT_POSITION_CONTAINS** is **0x08**.

Node.DOCUMENT_POSITION_CONTAINED_BY

The value of the constant **Node.DOCUMENT_POSITION_CONTAINED_BY** is **0x10**.

Node.DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC

The value of the constant **Node.DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC** is **0x20**.

Objects that implement the **Node** interface:

Properties of objects that implement the **Node** interface:

nodeName
: This read-only property is a **String**.

nodeValue
: This property is a **String**, can raise an object that implements the **DOMException** interface on setting and can raise an object that implements the **DOMException** interface on retrieval.

nodeType
: This read-only property is a **Number**.

parentNode
: This read-only property is an object that implements the **Node** interface.

childNodes
: This read-only property is an object that implements the **NodeList** interface.

firstChild
: This read-only property is an object that implements the **Node** interface.

lastChild
: This read-only property is an object that implements the **Node** interface.

previousSibling
: This read-only property is an object that implements the **Node** interface.

nextSibling
: This read-only property is an object that implements the **Node** interface.

attributes
: This read-only property is an object that implements the **NamedNodeMap** interface.

ownerDocument
: This read-only property is an object that implements the **Document** interface.

namespaceURI
: This read-only property is a **String**.

prefix
: This property is a **String** and can raise an object that implements the **DOMException** interface on setting.

localName
: This read-only property is a **String**.

baseURI
: This read-only property is a **String**.

textContent
: This property is a **String**, can raise an object that implements the **DOMException** interface on setting and can raise an object that implements the **DOMException** interface on retrieval.

Functions of objects that implement the **Node** interface:

insertBefore(newChild, refChild)
: This function returns an object that implements the **Node** interface.
: The **newChild** parameter is an object that implements the **Node** interface.
: The **refChild** parameter is an object that implements the **Node** interface.
: This function can raise an object that implements the **DOMException** interface.

replaceChild(newChild, oldChild)
: This function returns an object that implements the **Node** interface.

The **newChild** parameter is an object that implements the **Node** interface.
The **oldChild** parameter is an object that implements the **Node** interface.
This function can raise an object that implements the **DOMException** interface.

removeChild(oldChild)
This function returns an object that implements the **Node** interface.
The **oldChild** parameter is an object that implements the **Node** interface.
This function can raise an object that implements the **DOMException** interface.

appendChild(newChild)
This function returns an object that implements the **Node** interface.
The **newChild** parameter is an object that implements the **Node** interface.
This function can raise an object that implements the **DOMException** interface.

hasChildNodes()
This function returns a **Boolean**.

cloneNode(deep)
This function returns an object that implements the **Node** interface.
The **deep** parameter is a **Boolean**.

normalize()
This function has no return value.

isSupported(feature, version)
This function returns a **Boolean**.
The **feature** parameter is a **String**.
The **version** parameter is a **String**.

hasAttributes()
This function returns a **Boolean**.

compareDocumentPosition(other)
This function returns a **Number**.
The **other** parameter is an object that implements the **Node** interface.
This function can raise an object that implements the **DOMException** interface.

isSameNode(other)
This function returns a **Boolean**.
The **other** parameter is an object that implements the **Node** interface.

lookupPrefix(namespaceURI)
This function returns a **String**.
The **namespaceURI** parameter is a **String**.

isDefaultNamespace(namespaceURI)
This function returns a **Boolean**.
The **namespaceURI** parameter is a **String**.

lookupNamespaceURI(prefix)
This function returns a **String**.
The **prefix** parameter is a **String**.

isEqualNode(arg)
This function returns a **Boolean**.
The **arg** parameter is an object that implements the **Node** interface.

getFeature(feature, version)
This function returns an object that implements the **Object** interface.
The **feature** parameter is a **String**.

The **version** parameter is a **String**.

setUserData(key, data, handler)

This function returns an object that implements the **any type** interface.

The **key** parameter is a **String**.

The **data** parameter is an object that implements the **any type** interface.

The **handler** parameter is an object that implements the **UserDataHandler** interface.

getUserData(key)

This function returns an object that implements the **any type** interface.

The **key** parameter is a **String**.

Objects that implement the **NodeList** interface:

Properties of objects that implement the **NodeList** interface:

length

This read-only property is a **Number**.

Functions of objects that implement the **NodeList** interface:

item(index)

This function returns an object that implements the **Node** interface.

The **index** parameter is a **Number**.

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** function with that index.

Objects that implement the **NamedNodeMap** interface:

Properties of objects that implement the **NamedNodeMap** interface:

length

This read-only property is a **Number**.

Functions of objects that implement the **NamedNodeMap** interface:

getNamedItem(name)

This function returns an object that implements the **Node** interface.

The **name** parameter is a **String**.

setNamedItem(arg)

This function returns an object that implements the **Node** interface.

The **arg** parameter is an object that implements the **Node** interface.

This function can raise an object that implements the **DOMException** interface.

removeNamedItem(name)

This function returns an object that implements the **Node** interface.

The **name** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

item(index)

This function returns an object that implements the **Node** interface.

The **index** parameter is a **Number**.

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** function with that index.

getNamedItemNS(namespaceURI, localName)

This function returns an object that implements the **Node** interface.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

setNamedItemNS(arg)

This function returns an object that implements the **Node** interface.

The **arg** parameter is an object that implements the **Node** interface.

This function can raise an object that implements the **DOMException** interface.

removeNamedItemNS(namespaceURI, localName)

This function returns an object that implements the **Node** interface.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

Objects that implement the **CharacterData** interface:

Objects that implement the **CharacterData** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **CharacterData** interface:

data

This property is a **String**, can raise an object that implements the **DOMException** interface on setting and can raise an object that implements the **DOMException** interface on retrieval.

length

This read-only property is a **Number**.

Functions of objects that implement the **CharacterData** interface:

substringData(offset, count)

This function returns a **String**.

The **offset** parameter is a **Number**.

The **count** parameter is a **Number**.

This function can raise an object that implements the **DOMException** interface.

appendData(arg)

This function has no return value.

The **arg** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

insertData(offset, arg)

This function has no return value.

The **offset** parameter is a **Number**.

The **arg** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

deleteData(offset, count)

This function has no return value.

The **offset** parameter is a **Number**.

The **count** parameter is a **Number**.

This function can raise an object that implements the **DOMException** interface.

replaceData(offset, count, arg)

This function has no return value.

The **offset** parameter is a **Number**.

The **count** parameter is a **Number**.

The **arg** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

Objects that implement the **Attr** interface:

Objects that implement the **Attr** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **Attr** interface:

name

This read-only property is a **String**.

specified

This read-only property is a **Boolean**.

value

This property is a **String** and can raise an object that implements the **DOMException** interface on setting.

ownerElement

This read-only property is an object that implements the **Element** interface.

schemaTypeInfo

This read-only property is an object that implements the **TypeInfo** interface.

isId

This read-only property is a **Boolean**.

Objects that implement the **Element** interface:

Objects that implement the **Element** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **Element** interface:

tagName

This read-only property is a **String**.

schemaTypeInfo

This read-only property is an object that implements the **TypeInfo** interface.

Functions of objects that implement the **Element** interface:

getAttribute(name)

This function returns a **String**.

The **name** parameter is a **String**.

setAttribute(name, value)

This function has no return value.

The **name** parameter is a **String**.

The **value** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

removeAttribute(name)

This function has no return value.

The **name** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

getAttributeNode(name)

This function returns an object that implements the **Attr** interface.

The **name** parameter is a **String**.

setAttributeNode(newAttr)

This function returns an object that implements the **Attr** interface.

The **newAttr** parameter is an object that implements the **Attr** interface.

This function can raise an object that implements the **DOMException** interface.

removeAttributeNode(oldAttr)

This function returns an object that implements the **Attr** interface.

The **oldAttr** parameter is an object that implements the **Attr** interface.

This function can raise an object that implements the **DOMException** interface.

getElementsByTagName(name)

This function returns an object that implements the **NodeList** interface.

The **name** parameter is a **String**.

getAttributeNS(namespaceURI, localName)

This function returns a **String**.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

setAttributeNS(namespaceURI, qualifiedName, value)

This function has no return value.

The **namespaceURI** parameter is a **String**.

The **qualifiedName** parameter is a **String**.

The **value** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

removeAttributeNS(namespaceURI, localName)

This function has no return value.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

getAttributeNodeNS(namespaceURI, localName)

This function returns an object that implements the **Attr** interface.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

setAttributeNodeNS(newAttr)

This function returns an object that implements the **Attr** interface.

The **newAttr** parameter is an object that implements the **Attr** interface.

This function can raise an object that implements the **DOMException** interface.

getElementsByTagNameNS(namespaceURI, localName)

This function returns an object that implements the **NodeList** interface.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

hasAttribute(name)

This function returns a **Boolean**.

The **name** parameter is a **String**.

hasAttributeNS(namespaceURI, localName)

This function returns a **Boolean**.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

setIdAttribute(name, isId)

This function has no return value.

The **name** parameter is a **String**.

The **isId** parameter is a **Boolean**.

This function can raise an object that implements the **DOMException** interface.

setIdAttributeNS(namespaceURI, localName, isId)

This function has no return value.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

The **isId** parameter is a **Boolean**.

This function can raise an object that implements the **DOMException** interface.

setIdAttributeNode(idAttr, isId)

This function has no return value.

The **idAttr** parameter is an object that implements the **Attr** interface.

The **isId** parameter is a **Boolean**.

This function can raise an object that implements the **DOMException** interface.

Objects that implement the **Text** interface:

Objects that implement the **Text** interface have all properties and functions of the **CharacterData** interface as well as the properties and functions defined below.

Properties of objects that implement the **Text** interface:

isElementContentWhitespace

This read-only property is a **Boolean**.

wholeText

This read-only property is a **String**.

Functions of objects that implement the **Text** interface:

splitText(offset)

This function returns an object that implements the **Text** interface.

The **offset** parameter is a **Number**.

This function can raise an object that implements the **DOMException** interface.

replaceWholeText(content)

This function returns an object that implements the **Text** interface.

The **content** parameter is a **String**.

This function can raise an object that implements the **DOMException** interface.

Objects that implement the **Comment** interface:

Objects that implement the **Comment** interface have all properties and functions of the **CharacterData** interface.

Properties of the **TypeInfo** Constructor function:

TypeInfo.DERIVATION_RESTRICTION

The value of the constant **TypeInfo.DERIVATION_RESTRICTION** is **0x00000001**.

TypeInfo.DERIVATION_EXTENSION

The value of the constant **TypeInfo.DERIVATION_EXTENSION** is **0x00000002**.

TypeInfo.DERIVATION_UNION

The value of the constant **TypeInfo.DERIVATION_UNION** is **0x00000004**.

TypeInfo.DERIVATION_LIST

The value of the constant **TypeInfo.DERIVATION_LIST** is **0x00000008**.

Objects that implement the **TypeInfo** interface:

Properties of objects that implement the **TypeInfo** interface:

typeName

This read-only property is a **String**.

typeNamespace

This read-only property is a **String**.

Functions of objects that implement the **TypeInfo** interface:

isDerivedFrom(typeNamespaceArg, typeNameArg, derivationMethod)

This function returns a **Boolean**.

The **typeNamespaceArg** parameter is a **String**.

The **typeNameArg** parameter is a **String**.

The **derivationMethod** parameter is a **Number**.

Properties of the **UserDataHandler** Constructor function:

UserDataHandler.NODE_CLONED

The value of the constant **UserDataHandler.NODE_CLONED** is **1**.

UserDataHandler.NODE_IMPORTED

The value of the constant **UserDataHandler.NODE_IMPORTED** is **2**.

UserDataHandler.NODE_DELETED

The value of the constant **UserDataHandler.NODE_DELETED** is **3**.

UserDataHandler.NODE_RENAMED

The value of the constant **UserDataHandler.NODE_RENAMED** is **4**.

UserDataHandler.NODE_ADOPTED

The value of the constant **UserDataHandler.NODE_ADOPTED** is **5**.

UserDataHandler function:

This function has no return value. The first parameter is a **Number**. The second parameter is a **String**. The third parameter is an object that implements the **any type** interface. The fourth parameter is an object that implements the **Node** interface. The fifth parameter is an object that implements the **Node** interface.

Properties of the **DOMError** Constructor function:

DOMError.SEVERITY_WARNING

The value of the constant **DOMError.SEVERITY_WARNING** is **1**.

DOMError.SEVERITY_ERROR

The value of the constant **DOMError.SEVERITY_ERROR** is **2**.

DOMError.SEVERITY_FATAL_ERROR

The value of the constant **DOMError.SEVERITY_FATAL_ERROR** is **3**.

Objects that implement the **DOMError** interface:

Properties of objects that implement the **DOMError** interface:

severity

This read-only property is a **Number**.

message

This read-only property is a **String**.

type

This read-only property is a **String**.

relatedException

This read-only property is an object that implements the **Object** interface.

relatedData
This read-only property is an object that implements the **Object** interface.
location
This read-only property is an object that implements the **DOMLocator** interface.

DOMErrorHandler function:
This function returns a **Boolean**. The parameter is an object that implements the **DOMError** interface.

Objects that implement the **DOMLocator** interface:
Properties of objects that implement the **DOMLocator** interface:
lineNumber
This read-only property is a **Number**.
columnNumber
This read-only property is a **Number**.
byteOffset
This read-only property is a **Number**.
utf16Offset
This read-only property is a **Number**.
relatedNode
This read-only property is an object that implements the **Node** interface.
uri
This read-only property is a **String**.

Objects that implement the **DOMConfiguration** interface:
Properties of objects that implement the **DOMConfiguration** interface:
parameterNames
This read-only property is an object that implements the **DOMStringList** interface.
Functions of objects that implement the **DOMConfiguration** interface:
setParameter(name, value)
This function has no return value.
The **name** parameter is a **String**.
The **value** parameter is an object that implements the **any type** interface.
This function can raise an object that implements the **DOMException** interface.
getParameter(name)
This function returns an object that implements the **any type** interface.
The **name** parameter is a **String**.
This function can raise an object that implements the **DOMException** interface.
canSetParameter(name, value)
This function returns a **Boolean**.
The **name** parameter is a **String**.
The **value** parameter is an object that implements the **any type** interface.

Objects that implement the **CDATASection** interface:
Objects that implement the **CDATASection** interface have all properties and functions of the **Text** interface.

Objects that implement the **DocumentType** interface:
Objects that implement the **DocumentType** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **DocumentType** interface:

name

This read-only property is a **String**.

entities

This read-only property is an object that implements the **NamedNodeMap** interface.

notations

This read-only property is an object that implements the **NamedNodeMap** interface.

publicId

This read-only property is a **String**.

systemId

This read-only property is a **String**.

internalSubset

This read-only property is a **String**.

Objects that implement the **Notation** interface:

Objects that implement the **Notation** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **Notation** interface:

publicId

This read-only property is a **String**.

systemId

This read-only property is a **String**.

Objects that implement the **Entity** interface:

Objects that implement the **Entity** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **Entity** interface:

publicId

This read-only property is a **String**.

systemId

This read-only property is a **String**.

notationName

This read-only property is a **String**.

inputEncoding

This read-only property is a **String**.

xmlEncoding

This read-only property is a **String**.

xmlVersion

This read-only property is a **String**.

Objects that implement the **EntityReference** interface:

Objects that implement the **EntityReference** interface have all properties and functions of the **Node** interface.

Objects that implement the **ProcessingInstruction** interface:

Objects that implement the **ProcessingInstruction** interface have all properties and functions of the **Node** interface as well as the properties and functions defined below.

Properties of objects that implement the **ProcessingInstruction** interface:

target

This read-only property is a **String**.

data

This property is a **String** and can raise an object that implements the **DOMException** interface on setting.

Note: In addition of having `DOMConfiguration` [p.106] parameters exposed to the application using the `setParameter` and `getParameter`, those parameters are also exposed as ECMAScript properties on the `DOMConfiguration` object. The name of the parameter is converted into a property name using a camel-case convention: the character ’-’ (HYPHEN-MINUS) is removed and the following character is being replaced by its uppercase equivalent.

Appendix I: Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including participants of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Andrew Watson (Object Management Group), Andy Heninger (IBM), Angel Diaz (IBM), Arnaud Le Hors (W3C and IBM), Ashok Malhotra (IBM and Microsoft), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Ezell (Hewlett-Packard Company), David Singer (IBM), Dimitris Dimitriadis (Improve AB and invited expert), Don Park (invited), Elena Litani (IBM), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Jeroen van Rotterdam (X-Hive Corporation), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape/AOL), Jon Ferraiolo (Adobe), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Lauren Wood (SoftQuad Software Inc., *former Chair*), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mary Brady (NIST), 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 and former Chair*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL, *Chair*), Rezaur Rahman (Intel), Rich Rollman (Microsoft), Rick Gessner (Netscape), Rick Jelliffe (invited), Rob Relyea (Microsoft), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tim Yu (Oracle), Tom Pixley (Netscape/AOL), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections (Please, keep bugging us with your issues!).

Many thanks to Andrew Clover, Petteri Stenius, Curt Arnold, Glenn A. Adams, Christopher Aillon, Scott Nichol, François Yergeau, Anjana Manian, Susan Lesch, and Jeffery B. Rancier for their review and comments of this document.

Special thanks to the DOM Conformance Test Suites contributors: Fred Drake, Mary Brady (NIST), Rick Rivello (NIST), Robert Clary (Netscape), with a special mention to Curt Arnold.

I.1 Production Systems

This specification was written in XML. The HTML, OMG IDL, Java and ECMAScript 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.

After DOM Level 1, 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.

Glossary

Editors:
Arnaud Le Hors, W3C
Robert S. Sutor, IBM Research (for DOM Level 1)

Some 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.24] . 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.

ancestor
An *ancestor* node of any node A is any node above A in a tree model, 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.

anonymous type name
An *anonymous type name* is an implementation-defined, globally unique qualified name provided by the processor for every anonymous type declared in a schema [p.208] .

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, where "below" means "away from the root."

document element

There is only one document element in a `Document` [p.41] . This element node is a child of the `Document` node. See *Well-Formed XML Documents* in XML [*XML 1.0*].

document order

There is an ordering, *document order*, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the document element [p.206] node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). The attribute nodes of an element occur after the element and before its children. The relative order of attribute nodes is implementation-dependent.

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 1.0*].

information item

An information item is an abstract representation of some component of an XML document. See the [*XML Information Set*] for details.

logically-adjacent text nodes

Logically-adjacent text nodes are `Text` [p.95] or `CDATASection` [p.114] nodes that can be visited sequentially in document order [p.206] or in reversed document order without entering, exiting, or passing over `Element` [p.85] , `Comment` [p.99] , or `ProcessingInstruction` [p.118] nodes.

hosting implementation

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

HTML

The HyperText Markup Language (*HTML*) is a simple markup language used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of applications. [*HTML 4.01*]

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 [*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 [*XML Namespaces*].

namespace URI

A *namespace URI* is a URI that identifies an XML namespace. This is called the namespace name in Namespaces in XML [*XML Namespaces*]. See also sections 1.3.2 "*DOM URIs*" and 1.3.3 "*XML Namespaces*" regarding URIs and namespace URIs handling and comparison in the DOM APIs.

namespace well-formed

A node is a *namespace well-formed* XML node if it is a well-formed [p.208] node, and follows the productions and namespace constraints. If [*XML 1.0*] is used, the constraints are defined in [*XML Namespaces*]. If [*XML 1.1*] is used, the constraints are defined in [*XML Namespaces 1.1*].

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.

partially valid

A node in a DOM tree is *partially valid* if it is well formed [p.208] (this part is for comments and processing instructions) and its immediate children are those expected by the content model. The node may be missing trailing required children yet still be considered *partially valid*.

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 [*XML Namespaces*].

read only node

A *read only 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 read only node can possibly be moved, when it is not itself contained in a read only node.

root node

The *root node* is a node that is not a child of any other node. All other nodes are children or other descendants of the root node.

schema
: A *schema* defines a set of structural and value constraints applicable to XML documents. Schemas can be expressed in schema languages, such as DTD, XML Schema, etc.

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

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

token
: An information item such as an XML Name which has been tokenized [p.208] .

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
: A node is a *well-formed* XML node if its serialized form, without doing any transformation during its serialization, matches its respective production in [*XML 1.0*] or [*XML 1.1*] (depending on the XML version in use) with all well-formedness constraints related to that production, and if the entities which are referenced within the node are also well-formed. If namespaces for XML are in use, the node must also be namespace well-formed [p.207] .

XML
: Extensible Markup Language (*XML*) is an extremely simple dialect of SGML which is completely described in this document. The goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML. [*XML 1.0*]

References

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

K.1 Normative References

[ECMAScript]
ECMAScript Language Specification, Third Edition. European Computer Manufacturers Association, Standard ECMA-262, December 1999. This version of the ECMAScript Language is available from http://www.ecma-international.org/.

[ISO/IEC 10646]
ISO/IEC 10646-2000 (E). Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization, 2000. See also International Organization for Standardization, available at http://www.iso.ch, for the latest version.

[Java]
The Java Language Specification, J. Gosling, B. Joy, and G. Steele, Authors. Addison-Wesley, September 1996. Available at http://java.sun.com/docs/books/jls

[OMG IDL]
"OMG IDL Syntax and Semantics" defined in *The Common Object Request Broker: Architecture and Specification, version 2*, Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm.

[Unicode]
The Unicode Standard, Version 4, ISBN 0-321-18578-1, as updated from time to time by the publication of new versions. The Unicode Consortium, 2000. See also Versions of the Unicode Standard, available at http://www.unicode.org/unicode/standard/versions, for latest version and additional information on versions of the standard and of the Unicode Character Database.

[XML 1.0]
Extensible Markup Language (XML) 1.0 (Third Edition), T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Editors. World Wide Web Consortium, 4 February 2004, revised 10 February 1998 and 6 October 2000. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2004/REC-xml-20040204. The latest version of XML 1.0 is available at http://www.w3.org/TR/REC-xml.

[XML 1.1]
XML 1.1, T. Bray, and al., Editors. World Wide Web Consortium, 4 February 2004. This version of the XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml11-20040204. The latest version of XML 1.1 is available at http://www.w3.org/TR/xml11.

[XML Base]
XML Base, J. Marsh, Editor. World Wide Web Consortium, June 2001. This version of the XML Base Recommendation is http://www.w3.org/TR/2001/REC-xmlbase-20010627. The latest version of XML Base is available at http://www.w3.org/TR/xmlbase.

[XML Information Set]
XML Information Set (Second Edition), J. Cowan and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004, revised 24 October 2001. This version of the XML Information Set

Recommendation is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version of XML Information Set is available at http://www.w3.org/TR/xml-infoset.

[XML Namespaces]

Namespaces in XML, T. Bray, D. Hollander, and A. Layman, Editors. World Wide Web Consortium, 14 January 1999. This version of the Namespaces in XML Recommendation is http://www.w3.org/TR/1999/REC-xml-names-19990114. The latest version of Namespaces in XML is available at http://www.w3.org/TR/REC-xml-names.

[XML Namespaces 1.1]

Namespaces in XML 1.1, T. Bray, D. Hollander, A. Layman, and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004. This version of the Namespaces in XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml-names11-20040204. The latest version of Namespaces in XML 1.1 is available at http://www.w3.org/TR/xml-names11/.

[XML Schema Part 1]

XML Schema Part 1: Structures, H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Part 1 Recommendation is http://www.w3.org/TR/2001/REC-xmlschema-1-20010502. The latest version of XML Schema Part 1 is available at http://www.w3.org/TR/xmlschema-1.

[XPointer]

XPointer Framework, P. Grosso, E. Maler, J. Marsh, and N. Walsh., Editors. World Wide Web Consortium, 25 March 2003. This version of the XPointer Framework Recommendation is http://www.w3.org/TR/2003/REC-xptr-framework-20030325/. The latest version of XPointer Framework is available at http://www.w3.org/TR/xptr-framework/.

K.2 Informative References

[Canonical XML]

Canonical XML Version 1.0, J. Boyer, Editor. World Wide Web Consortium, 15 March 2001. This version of the Canonical XML Recommendation is http://www.w3.org/TR/2001/REC-xml-c14n-20010315. The latest version of Canonical XML is available at http://www.w3.org/TR/xml-c14n.

[COM]

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

[CORBA]

The Common Object Request Broker: Architecture and Specification, version 2. Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm.

[DOM Level 1]

DOM Level 1 Specification, V. Apparao, et al., Editors. World Wide Web Consortium, 1 October 1998. This version of the DOM Level 1 Recommendation is http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001. The latest version of DOM Level 1 is available at http://www.w3.org/TR/REC-DOM-Level-1.

[DOM Level 2 Core]

Document Object Model Level 2 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, 13 November 2000. This version of the DOM Level 2 Core Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113. The latest version of DOM Level

2 Core is available at http://www.w3.org/TR/DOM-Level-2-Core.

[DOM Level 3 Events]

Document Object Model Level 3 Events Specification, P. Le Hégaret, T. Pixley, Editors. World Wide Web Consortium, November 2003. This version of the Document Object Model Level 3 Events specification is http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107. The latest version of Document Object Model Level 3 Events is available at http://www.w3.org/TR/DOM-Level-3-Events.

[DOM Level 3 Load and Save]

Document Object Model Level 3 Load and Save Specification, J. Stenback, A. Heninger, Editors. World Wide Web Consortium, 7 April 2004. This version of the DOM Level 3 Load and Save Recommendation is http://www.w3.org/TR/2004/REC-DOM-Level-3-LS-20040407. The latest version of DOM Level 3 Load and Save is available at http://www.w3.org/TR/DOM-Level-3-LS.

[DOM Level 2 HTML]

Document Object Model Level 2 HTML Specification, J. Stenback, et al., Editors. World Wide Web Consortium, 9 January 2003. This version of the Document Object Model Level 2 HTML Recommendation is http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109. The latest version of Document Object Model Level 2 HTML is available at http://www.w3.org/TR/DOM-Level-2-HTML.

[DOM Level 3 Validation]

Document Object Model Level 3 Validation Specification, B. Chang, J. Kesselman, R. Rahman, Editors. World Wide Web Consortium, 27 January 2003. This version of the DOM Level 3 Validation Recommendation is http://www.w3.org/TR/2004/REC-DOM-Level-3-Val-20040127/. The latest version of DOM Level 3 Validation is available at http://www.w3.org/TR/DOM-Level-3-Val.

[DOM Level 3 XPath]

Document Object Model Level 3 XPath Specification, R. Whitmer, Editor. World Wide Web Consortium, March 2003. This version of the Document Object Model Level 3 XPath specification is http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226. The latest version of Document Object Model Level 3 XPath is available at http://www.w3.org/TR/DOM-Level-3-XPath.

[HTML 4.01]

HTML 4.01 Specification, D. Raggett, A. Le Hors, and I. Jacobs, Editors. World Wide Web Consortium, 17 December 1997, revised 24 April 1998, revised 24 December 1999. This version of the HTML 4.01 Recommendation is http://www.w3.org/TR/1999/REC-html401-19991224. The latest version of HTML 4 is available at http://www.w3.org/TR/html4.

[Java IDL]

Java IDL. Sun Microsystems. Available at http://java.sun.com/products/jdk/idl/

[JavaScript]

JavaScript Resources. Netscape Communications Corporation. Available at http://devedge.netscape.com/central/javascript/

[JScript]

JScript Resources. Microsoft. Available at http://msdn.microsoft.com/library/en-us/script56/html/js56jslrfjscriptlanguagereference.asp

[MathML 2.0]

Mathematical Markup Language (MathML) Version 2.0 (Second Edition), D. Carlisle, P. Ion, R. Miner, N. Poppelier, Editors. World Wide Web Consortium, 21 October 2001, revised 21 February 2001. This version of the Math 2.0 Recommendation is

http://www.w3.org/TR/2003/REC-MathML2-20031021. The latest version of MathML 2.0 is available at http://www.w3.org/TR/MathML2.

[MIDL]

MIDL Language Reference. Microsoft. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_language_reference.asp.

[IETF RFC 2396]

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

[SAX]

Simple API for XML, D. Megginson and D. Brownell, Maintainers. Available at http://www.saxproject.org/.

[SVG 1.1]

Scalable Vector Graphics (SVG) 1.1 Specification, J. Ferraiolo, 藤沢 淳 (FUJISAWA Jun), and D. Jackson, Editors. World Wide Web Consortium, 14 January 2003. This version of the SVG 1.1 Recommendation is http://www.w3.org/TR/2003/REC-SVG11-20030114/. The latest version of SVG 1.1 is available at http://www.w3.org/TR/SVG.

[XPath 1.0]

XML Path Language (XPath) Version 1.0, J. Clark and S. DeRose, Editors. World Wide Web Consortium, 16 November 1999. This version of the XPath 1.0 Recommendation is http://www.w3.org/TR/1999/REC-xpath-19991116. The latest version of XPath 1.0 is available at http://www.w3.org/TR/xpath.

Index