Document Object Model (DOM) Level 3 Core Specification

Version 1.0

W3C Working Draft 14 January 2002

This version:
http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020114
(PostScript file , PDF file , plain text , ZIP file , single HTML file)
Latest version:
http://www.w3.org/TR/DOM-Level-3-Core
Previous version:
http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913

Editors:
Arnaud Le Hors, *IBM*
Philippe Le Hégaret, *W3C, WG Chair*
Gavin Nicol, *Inso EPS (for DOM Level 1)*
Lauren Wood, *SoftQuad, Inc. (WG Chair emerata, for DOM Level 1 and 2)*
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)*

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].

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This document contains the Document Object Model Level 3 Core specification.

This is a Working Draft for review by W3C members and other interested parties.

It is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". This is work in progress and does not imply endorsement by, or the consensus of, either W3C or members of the DOM Working Group.

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

This document has been produced as part of the W3C DOM Activity. The authors of this document are the DOM Working Group members.

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

Table of contents

Expanded Table of Contents

Copyright Notice

W3C Document Copyright Notice and License

W3C Software Copyright Notice and License

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

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.147]) for valid *HTML* [p.148] and well-formed *XML* [p.150] 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.148] 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.147] . The DOM is designed to be used with any programming language. In order to provide a precise, language-independent specification of the DOM interfaces, we have chosen to define the specifications in Object Management Group (OMG) IDL [OMGIDL], as defined in the CORBA 2.3.1 specification [CORBA]. In addition to the OMG IDL specification, we provide *language bindings* [p.148] for Java [Java] and ECMAScript [ECMAScript] (an industry-standard scripting language based on JavaScript [JavaScript] and JScript [JScript]).

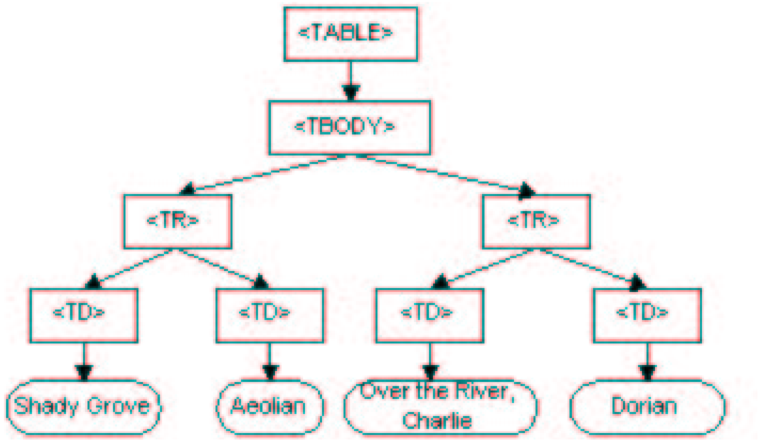
Note: OMG IDL is used only as a language-independent and implementation-neutral way to specify *interfaces* [p.148] . Various other IDLs could have been used ([COM], [Java IDL], [MIDL], ...). In general, IDLs are designed for specific computing environments. The Document Object Model can be implemented 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.147] for documents. It is based on an object structure that closely resembles the structure of the documents it *models* [p.149] . For instance, consider this table, taken from an HTML document:

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

A graphical representation of the DOM of the example table is:



graphical representation of the DOM of the example table

In the DOM, documents have a logical structure which is very much like a tree; to be more precise, which is like a "forest" or "grove", which can contain more than one tree. Each document contains zero or one doctype nodes, one 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.149] " 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.147] , not by an object model. In an abstract *data model* [p.147] , 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.149] 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.147] 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 "&amp;" 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 "&amp;" 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.15] .

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.148] 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.

Conformance

This section explains the different levels of conformance to DOM Level 3. DOM Level 3 consists of ? 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 [p.22]). 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.22] .

XML module
Defines the feature *"XML"* [p.90] .

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].

Mutation Events module
defines the feature *"MutationEvents"* 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 Abstract Schemas and Load and Save].

Abstract Schemas Editing module
defines the feature *"AS-EDIT"* in [DOM Level 3 Abstract Schemas and Load and Save].

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

A DOM implementation must not return `true` to the `hasFeature(feature, version)` *method* [p.149] of the `DOMImplementation` [p.25] 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)

1.1. Overview of the DOM Core Interfaces

This section defines a set of objects and interfaces for accessing and manipulating document objects. The functionality specified in this section (the *Core* functionality) is sufficient to allow software developers and web script authors to access and manipulate parsed HTML and XML content inside conforming products. The DOM Core *API* [p.147] also allows creation and population of a `Document` [p.29] object using only DOM API calls; loading a `Document` and saving it persistently is left to the product that implements the DOM API.

1.1.1. The DOM Structure Model

The DOM presents documents as a hierarchy of `Node` [p.49] objects that also implement other, more specialized interfaces. Some types of nodes may have *child* [p.147] 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.29] -- `Element` [p.77] (maximum of one), `ProcessingInstruction` [p.95] , `Comment` [p.87] , `DocumentType` [p.91] (maximum of one)
- `DocumentFragment` [p.28] -- `Element` [p.77] , `ProcessingInstruction` [p.95] , `Comment` [p.87] , `Text` [p.85] , `CDATASection` [p.90] , `EntityReference` [p.95]
- `DocumentType` [p.91] -- no children
- `EntityReference` [p.95] -- `Element` [p.77] , `ProcessingInstruction` [p.95] , `Comment` [p.87] , `Text` [p.85] , `CDATASection` [p.90] , `EntityReference`
- `Element` [p.77] -- `Element`, `Text` [p.85] , `Comment` [p.87] , `ProcessingInstruction` [p.95] , `CDATASection` [p.90] , `EntityReference` [p.95]
- `Attr` [p.75] -- `Text` [p.85] , `EntityReference` [p.95]
- `ProcessingInstruction` [p.95] -- no children
- `Comment` [p.87] -- no children
- `Text` [p.85] -- no children
- `CDATASection` [p.90] -- no children
- `Entity` [p.93] -- `Element` [p.77] , `ProcessingInstruction` [p.95] , `Comment` [p.87] , `Text` [p.85] , `CDATASection` [p.90] , `EntityReference` [p.95]
- `Notation` [p.93] -- no children

The DOM also specifies a `NodeList` [p.66] interface to handle ordered lists of `Nodes` [p.49] , such as the children of a `Node` [p.49] , or the *elements* [p.148] returned by the `getElementsByTagName` method of the `Element` [p.77] interface, and also a `NamedNodeMap` [p.67] interface to handle unordered sets of nodes referenced by their name attribute, such as the attributes of an `Element`. `NodeList` [p.66] and `NamedNodeMap` [p.67] 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.77] , then subsequently adds more children to that *element* [p.148] (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.49] in the tree are reflected in all references to that `Node` in `NodeList` and `NamedNodeMap` objects.

Finally, the interfaces `Text` [p.85] , `Comment` [p.87] , and `CDATASection` [p.90] all inherit from the `CharacterData` [p.71] 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.29] 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.148] and Java) require any memory management methods, but DOM bindings for other languages (especially C or C++) may require such support. These extensions will be the responsibility of those adapting the DOM API to a specific language, not the DOM Working Group.

1.1.3. Naming Conventions

While it would be nice to have attribute and method names that are short, informative, internally consistent, and familiar to users of similar APIs, the names also should not clash with the names in legacy APIs supported by DOM implementations. Furthermore, both OMG IDL and `ECMAScript` have significant limitations in their ability to disambiguate names from different namespaces that make it difficult to avoid naming conflicts with short, familiar names. So, 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.147] present two somewhat different sets of interfaces to an XML/HTML document: one presenting an "object oriented" approach with a hierarchy of *inheritance* [p.148] , and a "simplified" view that allows all manipulation to be done via the `Node` [p.49] interface without requiring casts (in Java and other C-like languages) or query interface calls in *COM* [p.147] 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.148] 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.147] .

In practice, this means that there is a certain amount of redundancy in the *API* [p.147] . The Working Group considers the "*inheritance* [p.148] " approach the primary view of the API, and the full set of functionality on `Node` [p.49] to be "extra" functionality that users may employ, but that does not eliminate the need for methods on other interfaces that an object-oriented analysis would dictate. (Of course, when the O-O analysis yields an attribute or method that is identical to one on the `Node` interface, we don’t specify a completely redundant one.) Thus, even though there is a generic `nodeName` attribute on the `Node` interface, there is still a `tagName` attribute on the `Element` [p.77] 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.147] must satisfy.

1.1.5. The `DOMString` type

To ensure interoperability, the DOM specifies the following:

Type Definition *DOMString*

A `DOMString` [p.17] is a sequence of *16-bit units* [p.147] .

IDL Definition

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

Applications must encode `DOMString` [p.17] using UTF-16 (defined in [Unicode 3.0] and Amendment 1 of [ISO/IEC 10646]).

The UTF-16 encoding was chosen because of its widespread industry practice. Note that for both HTML and XML, the document character set (and therefore the notation of numeric character references) is based on UCS [ISO/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.17] (a high surrogate and a low surrogate).

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

Note: As of August 2000, the OMG IDL specification ([OMGIDL]) included a `wstring` type. However, that definition did not meet the interoperability criteria of the DOM *API* [p.147] since it relied on negotiation to decide the width and encoding of a character.

1.1.6. The `DOMTimeStamp` type

To ensure interoperability, the DOM specifies the following:

Type Definition *DOMTimeStamp*

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

IDL Definition

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

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

1.1.7. The `DOMKeyObject` type

To ensure interoperability, the DOM specifies the following:

Type Definition *DOMKeyObject*

A `DOMKeyObject` [p.18] represents a reference to an application object.

IDL Definition

```
typedef Object DOMKeyObject;
```

Note: Even though the DOM uses the type `DOMKeyObject` [p.18] , bindings may use different types. For example, in Java `DOMKeyObject` is bound to the `Object` type, while in ECMAScript `DOMKeyObject` is bound to `any type`.

Issue DOMKeyObject-1:
What does DOMKeyObject map to in ECMAScript?
Resolution: "any type"

1.1.8. The `DOMObject` type

To ensure interoperability, the DOM specifies the following:

Type Definition *DOMObject*

A `DOMObject` [p.19] represents a reference to an application object.

IDL Definition

```
typedef Object DOMObject;
```

Note: Even though the DOM uses the type `DOMObject` [p.19] , bindings may use different types. For example, in Java and ECMAScript `DOMObject` is bound to the `Object` type.

1.1.9. String comparisons in the DOM

The DOM has many interfaces that imply string matching. HTML processors generally assume an uppercase (less often, lowercase) normalization of names for such things as *elements* [p.148] , while XML is explicitly case sensitive. For the purposes of the DOM, string matching is performed purely by binary *comparison* [p.149] of the *16-bit units* [p.147] of the `DOMString` [p.17] . In addition, the DOM assumes that any case normalizations take place in the processor, *before* the DOM structures are built.

The W3C Text normalization, as defined in [CharModel], is assumed to happen at serialization time. The DOM Level 3 Load and Save module [DOM Level 3 Abstract Schemas and Load and Save] provides a serialization mechanism (see the `DOMWriter` interface, section 2.3.1) and defines the `"ls-normalize-characters"` to assure that text is serialized in the W3C Text Normalization form. Other serialization mechanisms built on top of the DOM Level 3 Core also have to assure that text is serialized in the W3C Text Normalization form.

(***ED:*** We need to review the case sensitivity of methods and attributes and how it fits with XML and HTML. Current wording is not clear at all ...)

1.1.10. XML Namespaces

The DOM Level 2 (and higher) supports XML namespaces [XML Namespaces] by augmenting several interfaces of the DOM Level 1 Core to allow creating and manipulating *elements* [p.148] and attributes associated to a namespace.

As far as the DOM is concerned, special attributes used for declaring *XML namespaces* [p.150] are still exposed and can be manipulated just like any other attribute. However, nodes are permanently bound to *namespace URIs* [p.149] 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.149] or namespace URI. Similarly, creating a node with a namespace prefix and namespace URI, or changing the namespace prefix of a node, does not result in any addition, removal, or modification of any special attributes for declaring the appropriate XML namespaces. Namespace validation is not enforced; the DOM application is responsible. In particular, since the mapping between prefixes and namespace URIs is not enforced, in general, the resulting document cannot be serialized naively. For example, applications may have to declare every namespace in use when serializing a document.

DOM Level 2 (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.149] . How relative namespace URI references are treated is undefined. To ensure interoperability only absolute namespace URI references (i.e., URI references beginning with a scheme name and a colon) should be used. Note that because the DOM does no lexical checking, the empty string will be treated as a real namespace URI in DOM Level 2 methods. Applications must use the value `null` as the namespaceURI parameter for methods if they wish to have no namespace.

Note: In the DOM, all namespace declaration attributes are *by definition* bound to the namespace URI: "http://www.w3.org/2000/xmlns/". These are the attributes whose *namespace prefix* [p.149] or *qualified name* [p.149] is "xmlns". Although, at the time of writing, this is not part of the XML Namespaces specification [XML Namespaces], it is planned to be incorporated in a future revision.

In a document with no namespaces, the *child* [p.147] list of an `EntityReference` [p.95] node is always the same as that of the corresponding `Entity` [p.93] . This is not true in a document where an entity contains unbound *namespace prefixes* [p.149] . In such a case, the *descendants* [p.147] of the corresponding `EntityReference` nodes may be bound to different *namespace URIs* [p.149] , depending on where the entity references are. Also, because, in the DOM, nodes always remain bound to the same namespace URI, moving such `EntityReference` nodes can lead to documents that cannot be serialized. This is also true when the DOM Level 1 method `createEntityReference` of the `Document` [p.29] interface is used to create entity references that correspond to such entities, since the *descendants* [p.147] of the returned `EntityReference` are unbound. The DOM Level 2 does not support any mechanism to resolve namespace prefixes. For all of these reasons, use of such entities and entity references should be avoided or used with extreme care. A future Level of the DOM may include some additional support for handling these.

The new methods, such as `createElementNS` and `createAttributeNS` of the `Document` [p.29] interface, are meant to be used by namespace aware applications. Simple applications that do not use namespaces can use the DOM Level 1 methods, such as `createElement` and `createAttribute`. Elements and attributes created in this way do not have any namespace prefix, namespace URI, or local name.

Note: Given that the property [in-scope namespaces] defined in [XML Information set] is not accessible from DOM Level 3 Core, the properties [prefix] and [namespace name] defined by the Namespace Information Item in [XML Information set] are not accessible from DOM Level 3 Core. However, [DOM

Level 3 XPath] does provide a way to access them.

Note: DOM Level 1 methods are namespace ignorant. Therefore, while it is safe to use these methods when not dealing with namespaces, using them and the new ones at the same time should be avoided. DOM Level 1 methods solely identify attribute nodes by their `nodeName`. On the contrary, the DOM Level 2 methods related to namespaces, identify attribute nodes by their `namespaceURI` and `localName`. Because of this fundamental difference, mixing both sets of methods can lead to unpredictable results. In particular, using `setAttributeNS`, an *element* [p.148] may have two attributes (or more) that have the same `nodeName`, but different `namespaceURI`s. Calling `getAttribute` with that `nodeName` could then return any of those attributes. The result depends on the implementation. Similarly, using `setAttributeNode`, one can set two attributes (or more) that have different `nodeNames` but the same `prefix` and `namespaceURI`. In this case `getAttributeNodeNS` will return either attribute, in an implementation dependent manner. The only guarantee in such cases is that all methods that access a named item by its `nodeName` will access the same item, and all methods which access a node by its URI and local name will access the same node. For instance, `setAttribute` and `setAttributeNS` affect the node that `getAttribute` and `getAttributeNS`, respectively, return.

1.1.11. 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 their users. For example, the MathML [MathML 2.0] and SVG [SVG 1.0] specifications are developing 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 XML Namespaces Recommendation 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.29] 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.

1.1.12. 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 an implementation, 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.117] and ECMAScript Language Binding [p.131] for specifics.

In all cases, though, the `DOMImplementationRegistry` provides a `getDOMImplementation` method accepting a features string, which is passed to every known `DOMImplementationSource` [p.25] until a suitable `DOMImplementation` [p.25] is found and returned. This method is the same as the one found on the `DOMImplementationSource` interface defined below.

Any number of `DOMImplementationSource` [p.25] objects can be registered. A source may return one or more `DOMImplementation` [p.25] singletons or construct new `DOMImplementation` objects, depending upon whether the requested features require specialized state in the `DOMImplementation` object.

Issue Level-3-Bootstrap-1:
    Is this not generic enough?
    Resolution: Yes. (F2F 31 Jul 2001)
Issue Level-3-Bootstrap-2:
    Should the method `getDOMImplementation` be called `byFeature` instead?
    Resolution: No. (F2F 31 Jul 2001)

1.2. Fundamental Interfaces

The interfaces within this section are considered *fundamental*, and must be fully implemented by all conforming implementations of the DOM, including all HTML DOM implementations [DOM Level 1], unless otherwise specified.

(***ED:*** change link to DOM Level 2 HTML when available)

A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` [p.25] interface 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.66] .

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;
```

Definition group *ExceptionCode*

An integer indicating the type of error generated.

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

Defined Constants

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`VALIDATION_ERR`, introduced in **DOM Level 3**.
: If a call to a method such as `insertBefore` or `removeChild` would make the `Node` [p.49] invalid with respect to *"partial validity"* [p.149] , this exception would be raised and the operation would not be done. This code is used in [DOM Level 3 Abstract Schemas and Load and Save]. Refer to this specification for further information.

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

Interface *DOMImplementationSource*

This interface permits a DOM implementer to supply one or more implementations, based upon requested features. Each implemented `DOMImplementationSource` object is listed in the binding-specific list of available sources so that its `DOMImplementation` [p.25] objects are made available.

IDL Definition

```
interface DOMImplementationSource {
  DOMImplementation  getDOMImplementation(in DOMString features);
};
```

Methods

`getDOMImplementation`

A method to request a DOM implementation.

Parameters

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

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 Value

`DOMImplementation` [p.25]	An implementation that has the desired features, or `null` if this source has none.

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:
  DOMImplementation  getInterface(in DOMString feature);
};
```

Methods

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

Creates a DOM Document object of the specified type with its document element.

Parameters

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

The *namespace URI* [p.149] of the document element to create or `null`.

`qualifiedName` of type `DOMString`

The *qualified name* [p.149] of the document element to be created or `null`.

`doctype` of type `DocumentType` [p.91]

The type of document to be created or `null`.

When `doctype` is not `null`, its `Node.ownerDocument` [p.55] attribute is set to the document being created.

Return Value

`Document` [p.29]	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.23]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` 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 by DOM implementations which do not support the "XML" feature, if they choose not to support this method. **Note:** Other features introduced in the future, by the DOM WG or in extensions defined by other groups, may also demand support for this method; please consult the definition of the feature to see if it requires this method.

`createDocumentType` introduced in **DOM Level 2**

Creates an empty `DocumentType` [p.91] node. Entity declarations and notations are not made available. Entity reference expansions and default attribute additions do not occur. It is expected that a future version of the DOM will provide a way for populating a `DocumentType`.

Parameters

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

The *qualified name* [p.149] 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.91]	A new `DocumentType` node with `Node.ownerDocument` [p.55] set to `null`.

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed. NOT_SUPPORTED_ERR: May be raised by DOM implementations which do not support the `"XML"` feature, if they choose not to support this method. **Note:** Other features introduced in the future, by the DOM WG or in extensions defined by other groups, may also demand support for this method; please consult the definition of the feature to see if it requires this method.

`getInterface` introduced in **DOM Level 3**

This method makes available a `DOMImplementation`'s specialized interface (see Mixed DOM implementations [p.21]).

Parameters

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

The name of the feature requested (case-insensitive).

Return Value

`DOMImplementation` [p.25]	Returns an alternate `DOMImplementation` which implements the specialized APIs of the specified feature, if any, or `null` if there is no alternate `DOMImplementation` object which implements interfaces associated with that feature. Any alternate `DOMImplementation` returned by this method must delegate to the primary core `DOMImplementation` and not return results inconsistent with the primary `DOMImplementation`

No Exceptions

`hasFeature`

Test if the DOM implementation implements a specific feature.

Parameters

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

The name of the feature to test (case-insensitive). The values used by DOM features are defined throughout the DOM Level 3 specifications and listed in the Conformance [p.12] section. The name must be an *XML name* [p.150] . To avoid possible conflicts, as a convention, names referring to features defined outside the DOM specification should be made unique.

`version` of type `DOMString`

This is the version number of the feature to test. In Level 3, the string can be either "3.0", "2.0" or "1.0". If the version is `null` or empty string, supporting any version of the feature causes the method to return `true`.

Return Value

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

No Exceptions

Interface *DocumentFragment*

`DocumentFragment` is a "lightweight" or "minimal" `Document` [p.29] 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.49] -- 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.150] (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.85] 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.29] (or indeed any other `Node` [p.49] 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.149] ; the `DocumentFragment` acts as the parent of these nodes so that the user can use the standard methods from the `Node` interface, such as `insertBefore` and `appendChild`.

Note: The properties [notations] and [unparsed entities] defined by the Document Information Item in [XML Information set] are accessible through the `DocumentType` [p.91] interface. The property [all declarations processed] is not accessible through the DOM API.

IDL Definition

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

Interface *Document*

The `Document` interface represents the entire HTML or XML document. Conceptually, it is the *root* [p.149] 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.49] 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:
           attribute DOMString       actualEncoding;
  // Introduced in DOM Level 3:
           attribute DOMString       encoding;
  // Introduced in DOM Level 3:
           attribute boolean         standalone;
  // Introduced in DOM Level 3:
           attribute DOMString       version;
                                        // raises(DOMException) on setting

  // Introduced in DOM Level 3:
           attribute boolean         strictErrorChecking;
  // Introduced in DOM Level 3:
           attribute DOMErrorHandler errorHandler;
  // 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:
  void               normalizeDocument();
  // Introduced in DOM Level 3:
  boolean            canSetNormalizationFeature(in DOMString name, 
                                                in boolean state);
  // Introduced in DOM Level 3:
  void               setNormalizationFeature(in DOMString name, 
                                             in boolean state)
                                        raises(DOMException);
  // Introduced in DOM Level 3:
  boolean            getNormalizationFeature(in DOMString name)
                                        raises(DOMException);
  // Introduced in DOM Level 3:
  Node               renameNode(in Node n, 
                                in DOMString namespaceURI, 
                                in DOMString name)
                                        raises(DOMException);
};
```

Attributes

`actualEncoding` of type `DOMString` [p.17] , introduced in **DOM Level 3**
: An attribute specifying the actual encoding of this document. This is `null` otherwise.
This attribute represents the property [character encoding scheme] defined in [XML Information set].

`doctype` of type `DocumentType` [p.91] , readonly, modified in **DOM Level 3**
: The Document Type Declaration (see `DocumentType` [p.91]) associated with this document. For HTML documents as well as XML documents without a document type declaration this returns `null`. The DOM Level 2 does not support editing the Document Type Declaration.

`documentElement` of type `Element` [p.77] , readonly
: This is a *convenience* [p.147] attribute that allows direct access to the child node that is the *document element* [p.147] of the document.
This attribute represents the property [document element] defined in [XML Information set].

`documentURI` of type `DOMString` [p.17] , introduced in **DOM Level 3**
: The location of the document or `null` if undefined.
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.

`encoding` of type `DOMString` [p.17] , introduced in **DOM Level 3**
: An attribute specifying, as part of the XML declaration, the encoding of this document. This is `null` when unspecified.

`errorHandler` of type `DOMErrorHandler` [p.89] , introduced in **DOM Level 3**
: This attribute allows applications to specify a `DOMErrorHandler` [p.89] to be called in the event that an error is encountered while performing an operation on a document. Note that not all methods use this mechanism, see the description of each method for details.

`implementation` of type `DOMImplementation` [p.25] , readonly
: The `DOMImplementation` [p.25] object that handles this document. A DOM application may use objects from multiple implementations.

`standalone` of type `boolean`, introduced in **DOM Level 3**
: An attribute specifying, as part of the XML declaration, whether this document is standalone.
This attribute represents the property [standalone] defined in [XML Information set].

`strictErrorChecking` of type `boolean`, introduced in **DOM Level 3**
: An attribute specifying whether errors 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.23] . In case of error, the behavior is undefined. This attribute is `true` by defaults.

`version` of type `DOMString` [p.17] , introduced in **DOM Level 3**
: An attribute specifying, as part of the XML declaration, the version number of this document. This is `null` when unspecified.
This attribute represents the property [version] defined in [XML Information set].
Exceptions on setting

`DOMException` [p.23]	NOT_SUPPORTED_ERR: Raised if the version is set to a value that is not supported by this `Document`.

Methods

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

Changes the `ownerDocument` of a node, its children, as well as the attached attribute nodes if there are any. If the node has a parent it is first removed from its parent child list. This effectively allows moving a subtree from one document to another. 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.75] . 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.91] nodes cannot be adopted.

ELEMENT_NODE

Specified attribute nodes of the source element are adopted, and the generated `Attr` [p.75] nodes. 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.93] nodes cannot be adopted.

ENTITY_REFERENCE_NODE

Only the `EntityReference` [p.95] 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.93] nodes cannot be adopted.

PROCESSING_INSTRUCTION_NODE, TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE

These nodes can all be adopted. No specifics.

Issue adoptNode-1:

Should this method simply return null when it fails? How "exceptional" is failure for this method?

Resolution: Stick with raising exceptions only in exceptional circumstances, return null on failure (F2F 19 Jun 2000).

Issue adoptNode-2:

Can an entity node really be adopted?

Resolution: No, neither can Notation nodes (Telcon 13 Dec 2000).

Issue adoptNode-3:

Does this affect keys and hashCode’s of the adopted subtree nodes?
If so, what about readonly-ness of key and hashCode?
if not, would appendChild affect keys/hashCodes or would it generate exceptions if key’s are duplicate?

Resolution: Both keys and hashcodes have been dropped.

Parameters

`source` of type `Node` [p.49]

The node to move into this document.

Return Value

`Node` [p.49]	The adopted node, or `null` if this operation fails, such as when the source node comes from a different implementation.

Exceptions

`DOMException` [p.23]	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.

`canSetNormalizationFeature` introduced in **DOM Level 3**

Query whether setting a feature to a specific value is supported.
The feature name has the same form as a DOM `hasFeature` string.

Parameters

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

The name of the feature to check.

`state` of type `boolean`

The requested state of the feature (`true` or `false`).

Return Value

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

No Exceptions

`createAttribute`

Creates an `Attr` [p.75] of the given name. Note that the `Attr` instance can then be set on an `Element` [p.77] using the `setAttributeNode` method.
To create an attribute with a qualified name and namespace URI, use the `createAttributeNS` method.

Parameters

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

The name of the attribute.

Return Value

`Attr` [p.75]	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.23]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

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

Creates an attribute of the given qualified name and namespace URI.
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.17]
The *namespace URI* [p.149] of the attribute to create.

`qualifiedName` of type `DOMString`
The *qualified name* [p.149] of the attribute to instantiate.

Return Value

`Attr` [p.75]

A new `Attr` object with the following attributes:

Attribute	Value
`Node.nodeName` [p.55]	qualifiedName
`Node.namespaceURI` [p.54]	`namespaceURI`
`Node.prefix` [p.55]	prefix, extracted from `qualifiedName`, or `null` if there is no prefix
`Node.localName` [p.54]	*local name*, extracted from `qualifiedName`
`Attr.name` [p.76]	`qualifiedName`
`Node.nodeValue` [p.55]	the empty string

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character, per the XML 1.0 specification [XML 1.0]. 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", or if the `qualifiedName`, or its prefix, is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/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.90] node whose value is the specified string.

Parameters

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

The data for the `CDATASection` [p.90] contents.

Return Value

`CDATASection` [p.90]	The new `CDATASection` object.

Exceptions

`DOMException` [p.23]	NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createComment`

Creates a `Comment` [p.87] node given the specified string.

Parameters

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

The data for the node.

Return Value

`Comment` [p.87]	The new `Comment` object.

No Exceptions

`createDocumentFragment`

Creates an empty `DocumentFragment` [p.28] object.

Return Value

`DocumentFragment` [p.28]	A new `DocumentFragment`.

No Parameters

No Exceptions

`createElement`

Creates an element of the type specified. Note that the instance returned implements the `Element` [p.77] interface, so attributes can be specified directly on the returned object.
In addition, if there are known attributes with default values, `Attr` [p.75] nodes representing them are automatically created and attached to the element.
To create an element with a qualified name and namespace URI, use the `createElementNS` method.

Parameters

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

The name of the element type to instantiate. For XML, this is case-sensitive, otherwise it depends on the case-sentivity 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.77]	A new `Element` object with the `nodeName` attribute set to `tagName`, and `localName`, `prefix`, and `namespaceURI` set to `null`.

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

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

Creates an element of the given qualified name and namespace URI.
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.17]

The *namespace URI* [p.149] of the element to create.

`qualifiedName` of type `DOMString`

The *qualified name* [p.149] of the element type to instantiate.

Return Value

`Element` [p.77]	A new `Element` object with the following attributes:

Attribute	Value
`Node.nodeName` [p.55]	`qualifiedName`
`Node.namespaceURI` [p.54]	`namespaceURI`
`Node.prefix` [p.55]	prefix, extracted from `qualifiedName`, or `null` if there is no prefix
`Node.localName` [p.54]	*local name*, extracted from `qualifiedName`
`Element.tagName` [p.78]	`qualifiedName`

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character, per the XML 1.0 specification [XML 1.0]. 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`, 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]. 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.95] 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.93] node.

Note: If any descendant of the `Entity` [p.93] node has an unbound *namespace prefix* [p.149] , the corresponding descendant of the created `EntityReference` [p.95] node is also unbound; (its `namespaceURI` is `null`). The DOM Level 2 does not support any mechanism to resolve namespace prefixes.

Parameters
`name` of type `DOMString` [p.17]
The name of the entity to reference.
Return Value

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

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createProcessingInstruction`
Creates a `ProcessingInstruction` [p.95] node given the specified name and data strings.
Parameters
`target` of type `DOMString` [p.17]
The target part of the processing instruction.
`data` of type `DOMString`
The data for the node.
Return Value

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

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified target contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

`createTextNode`
Creates a `Text` [p.85] node given the specified string.
Parameters
`data` of type `DOMString` [p.17]
The data for the node.
Return Value

`Text` [p.85]	The new `Text` object.

No Exceptions

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

Returns the `Element` [p.77] whose `ID` is given by `elementId`. If no such element exists, returns `null`. Behavior is not defined if more than one element has this `ID`.

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

Parameters

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

The unique `id` value for an element.

Return Value

`Element` [p.77]	The matching element.

No Exceptions

`getElementsByTagName`

Returns a `NodeList` [p.66] of all the `Elements` [p.77] with a given tag name in *document order* [p.148] .

Parameters

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

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

Return Value

`NodeList` [p.66]	A new `NodeList` object containing all the matched `Elements` [p.77] .

No Exceptions

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

Returns a `NodeList` [p.66] of all the `Elements` [p.77] with a given *local name* [p.149] and namespace URI in *document order* [p.148] .

Parameters

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

The *namespace URI* [p.149] of the elements to match on. The special value "*" matches all namespaces.

`localName` of type `DOMString`

The *local name* [p.149] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.66]	A new `NodeList` object containing all the matched `Elements` [p.77] .

No Exceptions

`getNormalizationFeature` introduced in **DOM Level 3**

Look up the value of a feature.
The feature name has the same form as a DOM `hasFeature` string. The recognized features are the same as the ones defined for `setNormalizationFeature`.

Parameters

`name` of type `DOMString` [p.17]
The name of the feature to look up.

Return Value

`boolean`	The current state of the feature (`true` or `false`).

Exceptions

`DOMException` [p.23]	NOT_FOUND_ERR: Raised when the feature name is not recognized.

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

Imports a node from another document to this document. The returned node has no parent; (`parentNode` is `null`). The source node is not altered or removed from the original document; this method creates a new copy of the source node.
For all nodes, importing a node creates a node object owned by the importing document, with attribute values identical to the source node's `nodeName` and `nodeType`, plus the attributes related to namespaces (`prefix`, `localName`, and `namespaceURI`). As in the `cloneNode` operation, the source node is not altered. User data associated to the imported node is not carried over. However, if any `UserDataHandlers` [p.87] 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.75] . The *descendants* [p.147] 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.75] nodes; they always carry their children with them when imported.

DOCUMENT_FRAGMENT_NODE

If the `deep` option was set to `true`, the *descendants* [p.147] of the source `DocumentFragment` [p.28] 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.91] nodes cannot be imported.

ELEMENT_NODE

Specified attribute nodes of the source element are imported, and the generated `Attr` [p.75] nodes are attached to the generated `Element` [p.77] . 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.147] of the source element are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_NODE

`Entity` [p.93] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.91] 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.147] of the the source `Entity` [p.93] are recursively imported and the resulting nodes reassembled to form the corresponding subtree.

ENTITY_REFERENCE_NODE

Only the `EntityReference` [p.95] 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.93] nodes can be imported, however in the current release of the DOM the `DocumentType` [p.91] 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.71] 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.49]

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.75] , and `EntityReference` [p.95] nodes.

Return Value

`Node` [p.49]	The imported node that belongs to this `Document`.

Exceptions

`DOMException` [p.23]	NOT_SUPPORTED_ERR: Raised if the type of node being imported is not supported. INVALID_CHARACTER_ERR: Raised if one the imported names contain an illegal character. 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. The actual result depends on the features being set and governing what operations actually take place. See `setNormalizeFeature` for details.
Noticeably this method normalizes `Text` [p.85] nodes, makes the document "namespace wellformed", according to the algorithm described below in pseudo code, by adding missing namespace declaration attributes and adding or changing namespace prefixes, updates the replacement tree of `EntityReference` [p.95] nodes, normalizes attribute values, etc.
See Namespace normalization [p.99] for details on how namespace declaration attributes and prefixes are normalized.
Issue normalizeNS-1:
Any other name? Joe proposes normalizeNamespaces.
Resolution: normalizeDocument. (F2F 26 Sep 2001)
Issue normalizeNS-2:
How specific should this be? Should we not even specify that this should be done by walking down the tree?
Resolution: Very. See above.
Issue normalizeNS-3:
What does this do on attribute nodes?
Resolution: Doesn’t do anything (F2F 1 Aug 2000).
Issue normalizeNS-4:
How does it work with entity reference subtree which may be broken?
Resolution: This doesn’t affect entity references which are not visited in this operation (F2F 1 Aug 2000).
Issue normalizeNS-5:
Should this really be on Node?
Resolution: Yes, but this only works on Document, Element, and

DocumentFragment. On other types it is a no-op. (F2F 1 Aug 2000).
No. Now that it does much more than simply fixing namespaces it only makes sense on Document (F2F 26 Sep 2001).

Issue normalizeNS-6:
What happens with read-only nodes?

Issue normalizeNS-7:
What/how errors should be reported? Are there any?
Resolution: Through the error reporter.

Issue normalizeNS-8:
Should this be optional?
Resolution: No.

Issue normalizeNS-9:
What happens with regard to mutation events?

No Parameters
No Return Value
No Exceptions

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

Rename an existing 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. This only applies to nodes of type `ELEMENT_NODE` and `ATTRIBUTE_NODE`.
When a new node is created, the following operations are performed: the 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.77] 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 attach to the new node, the user data event `NODE_RENAMED` is fired.
When the node being renamed is an `Attr` [p.75] that is attached to an `Element` [p.77] , 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, when the implementation supports the feature "MutationEvents", each mutation operation involved in this method fires the appropriate event, and in the end the event `ElementNameChanged` or `AttributeNameChanged` is fired.

Issue renameNode-1:
Should this throw a HIERARCHY_REQUEST_ERR?

Parameters

`n` of type `Node` [p.49]
The node to rename.

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

`name` of type `DOMString`
The new qualified name.

Return Value

`Node` [p.49]	The renamed node. This is either the specified node or the new node that was created to replace the specified node.

Exceptions

`DOMException` [p.23]	NOT_SUPPORTED_ERR: Raised when the type of the specified node is neither `ELEMENT_NODE` nor `ATTRIBUTE_NODE`.

`setNormalizationFeature` introduced in **DOM Level 3**

Set the state of a feature.

Issue normalizationFeature-1:

Need to specify the list of features.

Feature names are valid XML names. Implementation specific features (extensions) should choose an implementation specific prefix to avoid name collisions. The following lists feature names that are recognized by all implementations. However, it is sometimes possible for a `Document` to recognize a feature but not to support setting its value. The following list of recognized features indicates the definitions of each feature state, if setting the state to `true` or `false` must be supported or is optional and, which state is the default one:

`"normalize-characters"`

`true`

[*optional*]

Perform the W3C Text Normalization of the characters [CharModel] in the document.

`false`

[*required*] (*default*)

Do not perform character normalization.

`"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.

`false`

[*required*]

Signal an error if a `CDATASection` [p.90] contains an unrepresentable character.

`"expand-entity-references"`

`true`

[*optional*]

Expand `EntityReference` [p.95] nodes when normalizing.

`false`

[*required*] (*default*)

Keep all `EntityReference` [p.95] nodes in document.

`"whitespace-in-element-content"`

`true`

[*required*] (*default*)
Keep all white spaces in the document.

`false`

[*optional*]
Discard white space in element content while normalizing. The implementation is expected to use the `isWhitespaceInElementContent` flag on `Text` [p.85] nodes to determine if a text node should be written out or not.

`"discard-default-content"`

`true`

[*required*] (*default*)
Use whatever information available to the implementation (i.e. XML schema, DTD, the `specified` flag on `Attr` [p.75] nodes, and so on) to decide what attributes and content should be discarded or not. Note that the `specified` flag on `Attr` nodes in itself is not always reliable, it is only reliable when it is set to `false` since the only case where it can be set to `false` is if the attribute was created by a Level 1 implementation.
Issue normalizationFeature-2:
How does exactly work? What’s the comment about level 1 implementations?

`false`

[*required*]
Keep all attributes and all content.

`"format-canonical"`

`true`

[*optional*]
Canonicalize the document according to the rules specified in [Canonical XML]. Setting this feature to true sets the feature "format-pretty-print" to false.

`false`

[*required*] (*default*)
Do not canonicalize the document.

`"format-pretty-print"`

`true`

[*optional*]
Format the document by adding whitespace to produce a pretty-printed, indented, human-readable form. The exact form of the transformations is not specified by this specification. Setting this feature to true sets the feature "format-canonical" to false.

`false`

[*required*] (*default*)
Do not pretty-print the document.

`"namespace-declarations"`

`true`

[*required*] (*default*)
Include namespace declaration attributes, specified or defaulted from the schema

or the DTD, in the document. See also the section *Declaring Namespaces* in [XML Namespaces].

`false`

[*optional*]

Discard all namespace declaration attributes. The Namespace prefixes are retained even if this feature is set to `false`.

`"validation"`

`true`

[*optional*]

Use the abstract schema to validate the document as it is being normalized. If validation errors are found the error handler is notified. Setting it to `true` also forces the `external-general-entities` and `external-parameter-entities` features to be `true`.) Also note that the `validate-if-schema` feature alters the validation behavior when this feature is set to `true`.

`false`

[*required*] (*default*)

Do not report validation errors.

`"external-parameter-entities"`

`true`

[*required*]

Load external parameter entities.

Issue normalizationFeature-3:

Doesn't really apply, does it? What does including them mean? Also, false can't be the default and be optional at the same time.

`false`

[*optional*] (*default*)

Do not load external parameter entities.

`"external-general-entities"`

`true`

[*required*] (*default*)

Include all external general (text) entities.

Issue normalizationFeature-4:

Doesn't really apply, does it? What does including them mean?

`false`

[*optional*]

Do not include external general entities.

`"external-dtd-subset"`

`true`

[*required*] (*default*)

Load the external DTD subset and also all external parameter entities.

Issue normalizationFeature-5:

Doesn't really apply, does it? What does loading mean here?

`false`

[*optional*]

Do not load the external DTD subset nor external parameter entities.

`"validate-if-schema"`

`true`

[*optional*]

When both this feature and validation are `true`, enable validation only if the document being processed has a schema (i.e. XML schema, DTD, any other type of schema, note that this is unrelated to the abstract schema specification). Documents without schemas are normalized without validation.

Issue normalizationFeature-6:

How does that interact with the notion of active AS?

`false`

[*required*] (*default*)

The validation feature alone controls whether the document is checked for validity. Documents without a schemas are not valid.

`"validate-against-dtd"`

`true`

[*optional*]

Prefer validation against the DTD over any other schema used with the document.

Issue normalizationFeature-7:

How does that interact with the notion of active AS?

`false`

[*required*] (*default*)

Let the implementation decide what to validate against if multiple types of schemas are in use.

`"datatype-normalization"`

`true`

[*required*]

Let the (non-DTD) validation process do its datatype normalization that is defined in the used schema language.

Issue normalizationFeature-8:

We should define "datatype normalization".

`false`

[*required*] (*default*)

Disable datatype normalization. The XML 1.0 attribute value normalization always occurs though.

`"create-entity-ref-nodes"`

`true`

[*required*] (*default*)

Create `EntityReference` [p.95] nodes in the document. It will also set `create-entity-nodes` to be `true`.

Issue normalizationFeature-9:

How does that interact with expand-entity-references? ALH suggests consolidating the two to a single feature called "entity-references" that is used both for load and save.

`false`

[*optional*]

Omit all `EntityReference` [p.95] nodes from the document, putting the entity expansions directly in their place. `Text` [p.85] nodes are into "normal" form. `EntityReference` nodes to non-defined entities are still created in the document.

`"create-entity-nodes"`

`true`

[*required*] (*default*)
Create `Entity` [p.93] nodes in the document.
Issue normalizationFeature-10:
How does that interact with expand-entity-references? ALH suggests renaming this one "entity-nodes", or simply "entities" for consistency.

`false`

[*optional*]
Omit all `entity` nodes from the document. It also sets `create-entity-ref-nodes` to `false`.

`"create-cdata-nodes"`

`true`

[*required*] (*default*)
Keep `CDATASection` [p.90] nodes the document.
Issue normalizationFeature-11:
Name does not work really well in this case. ALH suggests renaming this to "cdata-sections". It works for both load and save.

`false`

[*optional*]
Transform `CDATASection` [p.90] nodes in the document into `Text` [p.85] nodes. The new `Text` node is then combined with any adjacent `Text` node.

`"comments"`

`true`

[*required*] (*default*)
Keep `Comment` [p.87] nodes in the document.

`false`

[*required*]
Discard `Comment` [p.87] nodes in the Document.

`"load-as-infoset"`

`true`

[*optional*]
Only keep in the document the information defined in the XML Information Set [XML Information set].
This forces the following features to `false`: `namespace-declarations`, `validate-if-schema`, `create-entity-ref-nodes`, `create-entity-nodes`, `create-cdata-nodes`.
This forces the following features to `true`: `datatype-normalization`, `whitespace-in-element-content`, `comments`.
Other features are not changed unless explicity specified in the description of the features.
Note that querying this feature with `getFeature` returns `true` only if the

individual features specified above are appropriately set.

Issue normalizationFeature-12:

Name doesn't work well here. ALH suggests renaming this to limit-to-infoset or match-infoset, something like that.

`false`

Setting `load-as-infoset` to `false` has no effect.

Issue normalizationFeature-13:

Shouldn't we change this to setting the relevant options back to their default value?

Parameters

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

The name of the feature to set.

`state` of type `boolean`

The requested state of the feature (`true` or `false`).

Exceptions

`DOMException` [p.23]	NOT_SUPPORTED_ERR: Raised when the feature name is recognized but the requested value cannot be set. NOT_FOUND_ERR: Raised when the feature name is not recognized.

No Return Value

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.85] nodes may not have children, and adding children to such nodes results in a `DOMException` [p.23] 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.77] or `attributes` for a `Comment` [p.87]), 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);
Node               appendChild(in Node newChild)
                                      raises(DOMException);
boolean            hasChildNodes();
Node               cloneNode(in boolean deep);
// Modified in DOM Level 2:
void               normalize();
// Introduced in DOM Level 2:
boolean            isSupported(in DOMString feature,
                               in DOMString version);
// Introduced in DOM Level 2:
readonly attribute DOMString       namespaceURI;
// Introduced in DOM Level 2:
         attribute DOMString       prefix;
                                      // raises(DOMException) on setting

// Introduced in DOM Level 2:
readonly attribute DOMString       localName;
// Introduced in DOM Level 2:
boolean            hasAttributes();
// Introduced in DOM Level 3:
readonly attribute DOMString       baseURI;

// TreePosition
```

```
  const unsigned short      TREE_POSITION_PRECEDING        = 0x01;
  const unsigned short      TREE_POSITION_FOLLOWING        = 0x02;
  const unsigned short      TREE_POSITION_ANCESTOR         = 0x04;
  const unsigned short      TREE_POSITION_DESCENDANT       = 0x08;
  const unsigned short      TREE_POSITION_EQUIVALENT       = 0x10;
  const unsigned short      TREE_POSITION_SAME_NODE        = 0x20;
  const unsigned short      TREE_POSITION_DISCONNECTED     = 0x00;

  // Introduced in DOM Level 3:
  unsigned short     compareTreePosition(in Node other);
  // 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          lookupNamespacePrefix(in DOMString namespaceURI);
  // Introduced in DOM Level 3:
  DOMString          lookupNamespaceURI(in DOMString prefix);
  // Introduced in DOM Level 3:
  boolean            isEqualNode(in Node arg,
                                 in boolean deep);
  // Introduced in DOM Level 3:
  Node               getInterface(in DOMString feature);
  // Introduced in DOM Level 3:
  DOMKeyObject       setUserData(in DOMString key,
                                 in DOMKeyObject data,
                                 in UserDataHandler handler);
  // Introduced in DOM Level 3:
  DOMKeyObject       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.75] .

`CDATA_SECTION_NODE`
: The node is a `CDATASection` [p.90] .

`COMMENT_NODE`
: The node is a `Comment` [p.87] .

`DOCUMENT_FRAGMENT_NODE`
: The node is a `DocumentFragment` [p.28] .

`DOCUMENT_NODE`
: The node is a `Document` [p.29] .

`DOCUMENT_TYPE_NODE`
: The node is a `DocumentType` [p.91] .

`ELEMENT_NODE`
: The node is an `Element` [p.77] .

`ENTITY_NODE`
: The node is an `Entity` [p.93] .

`ENTITY_REFERENCE_NODE`
: The node is an `EntityReference` [p.95] .

`NOTATION_NODE`
: The node is a `Notation` [p.93] .

`PROCESSING_INSTRUCTION_NODE`
: The node is a `ProcessingInstruction` [p.95] .

`TEXT_NODE`
: The node is a `Text` [p.85] node.

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

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

Definition group *TreePosition*

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

Issue TreePosition-1:
Should we use fewer bits?
Resolution: No. Simpler that way.

Issue TreePosition-2:
How does a node compare to itself?
Resolution: SAME_NODE and EQUIVALENT. (F2F 26 Sep 2001)

Defined Constants

`TREE_POSITION_ANCESTOR`
The node is an ancestor of the reference node.

`TREE_POSITION_DESCENDANT`
The node is a descendant of the reference node.

`TREE_POSITION_DISCONNECTED`
The two nodes are disconnected, they do not have any common ancestor. This is the case of two nodes that are not in the same document.

`TREE_POSITION_EQUIVALENT`
The two nodes have an equivalent position. This is the case of two attributes that have the same `ownerElement`, and two nodes that are the same.

`TREE_POSITION_FOLLOWING`
The node follows the reference node.

`TREE_POSITION_PRECEDING`
The node precedes the reference node.

`TREE_POSITION_SAME_NODE`
The two nodes are the same. Two nodes that are the same have an equivalent position, though the reverse may not be true.

Attributes

`attributes` of type `NamedNodeMap` [p.67] , readonly
A `NamedNodeMap` [p.67] containing the attributes of this node (if it is an `Element` [p.77]) or `null` otherwise.
If no namespace declaration appear in the attributes, this attribute represents the property [attributes] defined in [XML Information set].

`baseURI` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 3**
The absolute base URI of this node or `null` if undefined. This value is computed according to [XML Base]. However, when the `Document` [p.29] 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.
When the node is an `Element` [p.77] , a `Document` [p.29] or a a `ProcessingInstruction` [p.95] , this attribute represents the properties [base URI] defined in [XML Information set]. When the node is a `Notation` [p.93] , an `Entity` [p.93] , or an `EntityReference` [p.95] , this attribute represents the properties [declaration base URI] in the [XML Information set].

Issue baseURI-1:
: How will this be affected by resolution of relative namespace URIs issue?
: **Resolution:** It's not.

Issue baseURI-2:
: Should this only be on Document, Element, ProcessingInstruction, Entity, and Notation nodes, according to the infoset? If not, what is it equal to on other nodes? Null? An empty string? I think it should be the parent's.
: **Resolution:** No.

Issue baseURI-3:
: Should this be read-only and computed or and actual read-write attribute?
: **Resolution:** Read-only and computed (F2F 19 Jun 2000 and teleconference 30 May 2001).

Issue baseURI-4:
: If the base HTML element is not yet attached to a document, does the insert change the Document.baseURI?
: **Resolution:** Yes. (F2F 26 Sep 2001)

`childNodes` of type `NodeList` [p.66] , readonly
: A `NodeList` [p.66] that contains all children of this node. If there are no children, this is a `NodeList` containing no nodes.
: When the node is a `Document` [p.29] , or an `Element` [p.77] , and if the `NodeList` [p.66] does not contain `EntityReference` [p.95] or `CDATASection` [p.90] nodes, this attribute represents the properties [children] defined in [XML Information set].

`firstChild` of type `Node` [p.49] , readonly
: The first child of this node. If there is no such node, this returns `null`.

`lastChild` of type `Node` [p.49] , readonly
: The last child of this node. If there is no such node, this returns `null`.

`localName` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
: Returns the local part of the *qualified name* [p.149] of this node.
: When the node is `Element` [p.77] , or `Attr` [p.75] , this attribute represents the properties [local name] defined in [XML Information set].
: 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.29] interface, this is always `null`.

`namespaceURI` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 2**
: The *namespace URI* [p.149] of this node, or `null` if it is unspecified.
: When the node is `Element` [p.77] , or `Attr` [p.75] , this attribute represents the properties [namespace name] defined in [XML Information set].
: This is not a computed value that is the result of a namespace lookup based on an examination of the namespace declarations in scope. It is merely the namespace URI given at creation time.
: For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` [p.29] interface, 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.49] , readonly
The node immediately following this node. If there is no such node, this returns `null`.

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

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

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

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.23]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.17] variable on the implementation platform.

`ownerDocument` of type `Document` [p.29] , readonly, modified in **DOM Level 2**
The `Document` [p.29] 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.91] which is not used with any `Document` yet, this is `null`.

`parentNode` of type `Node` [p.49] , readonly
The *parent* [p.149] of this node. All nodes, except `Attr` [p.75] , `Document` [p.29] , `DocumentFragment` [p.28] , `Entity` [p.93] , and `Notation` [p.93] 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`.
When the node is an `Element` [p.77] , a `ProcessingInstruction` [p.95] , an `EntityReference` [p.95] , a `CharacterData` [p.71] , a `Comment` [p.87] , or a `DocumentType` [p.91] , this attribute represents the properties [parent] defined in [XML Information set].

`prefix` of type `DOMString` [p.17] , introduced in **DOM Level 2**
The *namespace prefix* [p.149] of this node, or `null` if it is unspecified.
When the node is `Element` [p.77] , or `Attr` [p.75] , this attribute represents the properties [prefix] defined in [XML Information set].
Note that setting this attribute, when permitted, changes the `nodeName` attribute, which holds the *qualified name* [p.149] , as well as the `tagName` and `name` attributes of the `Element` [p.77] and `Attr` [p.75] interfaces, when applicable.
Note also that changing the prefix of an attribute that is known to have a default value, does not make a new attribute with the default value and the original prefix appear, since the

`namespaceURI` and `localName` do not change.
For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` [p.29] interface, this is always `null`.
Exceptions on setting

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified prefix contains an illegal character, per the XML 1.0 specification [XML 1.0]. 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.49] , readonly
The node immediately preceding this node. If there is no such node, this returns `null`.
`textContent` of type `DOMString` [p.17] , 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. When set, any possible children this node may have are removed and replaced by a single `Text` [p.85] 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, the returned string does not contain the element content whitespaces Fundamental Interfaces [p.85] . 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, 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
ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE, PROCESSING_INSTRUCTION_NODE	`nodeValue`
DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE	*null*

Issue textContent-1:
: Should any whitespace normalization be performed? MS' text property doesn't but what about "ignorable whitespace"?
: **Resolution:** Does not perform any whitespace normalization and ignores "ignorable whitespace".

Issue textContent-2:
: Should this be two methods instead?
: **Resolution:** No. Keep it a read write attribute.

Issue textContent-3:
: What about the name? MS uses text and innerText. text conflicts with HTML DOM.
: **Resolution:** Keep the current name, MS has a different name and different semantic.

Issue textContent-4:
: Should this be optional?
: **Resolution:** No.

Issue textContent-5:
: Setting the text property on a Document, Document Type, or Notation node is an error for MS. How do we expose it? Exception? Which one?
: **Resolution:** (teleconference 23 May 2001) consistency with nodeValue. Remove Document from the list.

Exceptions on setting

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.23]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.17] variable on the implementation platform.

Methods

`appendChild`

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

Parameters

`newChild` of type `Node` [p.49]

The node to add.

If it is a `DocumentFragment` [p.28] object, the entire contents of the document fragment are moved into the child list of this node

Return Value

`Node` [p.49]	The node added.

Exceptions

`DOMException` [p.23]	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.147] or this node itself. 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.

`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.87] 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.77] 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.85] node. Cloning an `Attribute` directly, as opposed to be cloned as part of an `Element` cloning operation, returns a specified attribute (`specified` is `true`). Cloning an `Attribute` always clones its children, since they represent its value, no matter whether this is a deep clone or not. Cloning an `EntityReference` [p.95] automatically constructs its subtree if a corresponding `Entity` [p.93] 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.95] clone are *readonly* [p.149] . In addition, clones of unspecified

`Attr` [p.75] nodes are specified. And, cloning `Document` [p.29] , `DocumentType` [p.91] , `Entity` [p.93] , and `Notation` [p.93] 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.77]).

Return Value

`Node` [p.49]	The duplicate node.

No Exceptions

`compareTreePosition` introduced in **DOM Level 3**

Compares a node with this node with regard to their position in the tree and according to the *document order* [p.148] . This order can be extended by module that define additional types of nodes.

Issue compareTreePosition-1:

Should this method be optional?

Resolution: No.

Issue compareTreePosition-2:

Need reference for namespace nodes.

Resolution: No, instead avoid referencing them directly.

Parameters

`other` of type `Node` [p.49]

The node to compare against this node.

Return Value

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

No Exceptions

`getInterface` introduced in **DOM Level 3**

This method makes available a `Node`'s specialized interface (see Mixed DOM implementations [p.21]).

Issue EDOM-isSupported:

What are the relations between Node.isSupported and Node3.getInterface?

Issue EDOM-getInterface-1:

Should we rename this method (and also DOMImplementation.getInterface?)?

Issue EDOM-getInterface-2:

getInterface can return a node that doesn't actually support the requested interface and will lead to a cast exception. Other solutions are returning null or throwing an exception.

Parameters

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

The name of the feature requested (case-insensitive).

Return Value

`Node` [p.49]	Returns an alternate `Node` which implements the specialized APIs of the specified feature, if any, or `null` if there is no alternate `Node` which implements interfaces associated with that feature. Any alternate `Node` returned by this method must delegate to the primary core `Node` and not return results inconsistent with the primary core `Node` such as `key`, `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.17]

The key the object is associated to.

Return Value

`DOMKeyObject` [p.18]	Returns the `DOMKeyObject` 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`	`true` if this node has any attributes, `false` otherwise.

No Parameters

No Exceptions

`hasChildNodes`

Returns whether this node has any children.

Return Value

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

No Parameters

No Exceptions

`insertBefore` 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.28] object, all of its children are inserted, in the same order, before `refChild`. If the `newChild` is already in the tree, it is first removed.

Parameters

`newChild` of type `Node` [p.49]
    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.49]	The node being inserted.

Exceptions

`DOMException` [p.23]	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.147] or this node itself, or if this node if of type `Document` [p.29] and the DOM application attempts to insert a second `DocumentType` [p.91] or `Element` [p.77] 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 if of type `Document` [p.29] , this exception might be raised if the DOM implementation doesn't support the insertion of a `DocumentType` [p.91] or `Element` [p.77] node.

`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.62] . 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`, `baseURI`. This is: they are both `null`, or they have the same length and are character for character identical.
- The `attributes` `NamedNodeMaps` [p.67] 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.66] are equal. This is: they are both `null`, or

they have the same length and contain equal nodes at the same index. This is true for `Attr` [p.75] nodes as for any other type of node. Note that normalization can affect equality; to avoid this, nodes should be normalized before being compared.

For two `DocumentType` [p.91] nodes to be equal, the following conditions must also be satisfied:

- The following string attributes are equal: `publicId`, `systemId`, `internalSubset`.
- The `entities` `NamedNodeMaps` [p.67] are equal.
- The `notations` `NamedNodeMaps` [p.67] are equal.

On the other hand, the following do not affect equality: the `ownerDocument` attribute, the `specified` attribute for `Attr` [p.75] nodes, the `isWhitespaceInElementContent` attribute for `Text` [p.85] nodes, as well as any user data or event listeners registered on the nodes.

Issue isEqualNode-1:

Should this be optional?

Resolution: No.

Parameters

`arg` of type `Node` [p.49]

The node to compare equality with.

`deep` of type `boolean`

If `true`, recursively compare the subtrees; if `false`, compare only the nodes themselves (and its attributes, if it is an `Element` [p.77]).

Return Value

`boolean`	If the nodes, and possibly subtrees are equal, `true` otherwise `false`.

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.

Issue isSameNode-1:

Do we really want to make this different from equals?

Resolution: Yes, change name from isIdentical to isSameNode. (Telcon 4 Jul 2000).

Issue isSameNode-2:

Is this really needed if we provide a unique key?

Resolution: Yes, because the key is only unique within a document. (F2F 2 Mar 2001).

Issue isSameNode-3:

Definition of ’sameness’ is needed.

Parameters

`other` of type `Node` [p.49]
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.

Parameters

`feature` of type `DOMString` [p.17]
The name of the feature to test. This is the same name which can be passed to the method `hasFeature` on `DOMImplementation` [p.25] .

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

Return Value

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

No Exceptions

`lookupNamespacePrefix` introduced in **DOM Level 3**

Look up the prefix associated to the given namespace URI, starting from this node.
See Namespace Prefix Lookup [p.102] for details on the algorithm used by this method.

Issue lookupNamespacePrefix-1:
Should this be optional?
Resolution: No.

Issue lookupNamespacePrefix-2:
How does the lookup work? Is it based on the prefix of the nodes, the namespace declaration attributes, or a combination of both?
Resolution: See Namespace Prefix Lookup [p.102] .

Parameters

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

Return Value

`DOMString` [p.17]	Returns the associated namespace prefix 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

`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.103] for details on the algorithm used by this method.

Issue lookupNamespaceURI-1:

Name? May need to change depending on ending of the relative namespace URI reference nightmare.

Resolution: No need.

Issue lookupNamespaceURI-2:

Should this be optional?

Resolution: No.

Issue lookupNamespaceURI-3:

How does the lookup work? Is it based on the namespaceURI of the nodes, the namespace declaration attributes, or a combination of both?

Resolution: See Namespace URI Lookup [p.103] .

Parameters

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

The prefix to look for.

Return Value

`DOMString` [p.17]	Returns the associated namespace URI or `null` if none is found.

No Exceptions

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

Puts all `Text` [p.85] nodes in the full depth of the sub-tree underneath this `Node`, including attribute nodes, into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates `Text` nodes, i.e., there are neither adjacent `Text` nodes nor empty `Text` nodes. This can be used to ensure that the DOM view of a document is the same as if it were saved and re-loaded, and is useful when operations (such as XPointer [XPointer] lookups) that depend on a particular document tree structure are to be used.

Note: In cases where the document contains `CDATASections` [p.90] , the normalize operation alone may not be sufficient, since XPointers do not differentiate between `Text` [p.85] nodes and `CDATASection` [p.90] 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.49]

The node being removed.

Return Value

`Node` [p.49]	The node removed.

Exceptions

`DOMException` [p.23]	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 if of type `Document` [p.29] , this exception might be raised if the DOM implementation doesn't support the removal of the `DocumentType` [p.91] child or the `Element` [p.77] 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.28] object, `oldChild` is replaced by all of the `DocumentFragment` children, which are inserted in the same order. If the `newChild` is already in the tree, it is first removed.

Parameters

`newChild` of type `Node` [p.49]
  The new node to put in the child list.

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

Return Value

`Node` [p.49]	The node replaced.

Exceptions

`DOMException` [p.23]	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.147] or this node itself. 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 if of type `Document` [p.29] , this exception might be raised if the DOM implementation doesn't support the replacement of the `DocumentType` [p.91] child or `Element` [p.77] 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.17]
The key to associate the object to.

`data` of type `DOMKeyObject` [p.18]
The object to associate to the given key, or `null` to remove any existing association to that key.

`handler` of type `UserDataHandler` [p.87]
The handler to associate to that key, or `null`.

Return Value

`DOMKeyObject` [p.18]	Returns the `DOMKeyObject` 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.16] .

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

IDL Definition

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

Attributes

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

Methods

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

Parameters

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

Return Value

`Node` [p.49]	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.66] ; `NamedNodeMaps` are not maintained in any particular order. Objects contained in an object implementing `NamedNodeMap` may also be accessed by an ordinal index, but this is simply to allow convenient enumeration of the contents of a `NamedNodeMap`, and does not imply that the DOM specifies an order to these Nodes.

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

IDL Definition

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

```
                                        raises(DOMException);
  // Introduced in DOM Level 2:
  Node               removeNamedItemNS(in DOMString namespaceURI,
                                       in DOMString localName)
                                        raises(DOMException);
};
```

Attributes

`length` of type `unsigned long`, readonly

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

Methods

`getNamedItem`

Retrieves a node specified by name.

Parameters

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

The `nodeName` of a node to retrieve.

Return Value

`Node` [p.49]	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.

Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.

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.17]

The *namespace URI* [p.149] of the node to retrieve.

`localName` of type `DOMString`

The *local name* [p.149] of the node to retrieve.

Return Value

`Node` [p.49]	A `Node` (of any type) with the specified local name and namespace URI, or `null` if they do not identify any node in this map.

No Exceptions

`item`

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

Parameters

`index` of type `unsigned long`

Index into this map.

Return Value

`Node` [p.49]	The node at the `index`th position in the map, or `null` if that is not a valid index.

No Exceptions

`removeNamedItem`

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

Parameters

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

The `nodeName` of the node to remove.

Return Value

`Node` [p.49]	The node removed from this map if a node with such a name exists.

Exceptions

`DOMException` [p.23]	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.49] interface. If so, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.

Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.

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.17]

The *namespace URI* [p.149] of the node to remove.

`localName` of type `DOMString`

The *local name* [p.149] of the node to remove.

Return Value

`Node` [p.49]	The node removed from this map if a node with such a local name and namespace URI exists.

Exceptions

`DOMException` [p.23]	NOT_FOUND_ERR: Raised if there is no node with the specified `namespaceURI` and `localName` in this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly.

`setNamedItem`

Adds a node using its `nodeName` attribute. If a node with that name is already present in this map, it is replaced by the new one.
As the `nodeName` attribute is used to derive the name which the node must be stored under, multiple nodes of certain types (those that have a "special" string value) cannot be stored as the names would clash. This is seen as preferable to allowing nodes to be aliased.

Parameters

`arg` of type `Node` [p.49]
A node to store in this map. The node will later be accessible using the value of its `nodeName` attribute.

Return Value

`Node` [p.49]	If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.23]	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.75] that is already an attribute of another `Element` [p.77] 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.
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.49]

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

Exceptions

`DOMException` [p.23]	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.75] that is already an attribute of another `Element` [p.77] 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: Always thrown if the current document does not support the `"XML"` feature, since namespaces were defined by XML.

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

As explained in the `DOMString` [p.17] interface, text strings in the DOM are represented in UTF-16, i.e. as a sequence of 16-bit units. In the following, the term *16-bit units* [p.147] is used whenever necessary to indicate that indexing on CharacterData is done in 16-bit units.

IDL Definition

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

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

Attributes

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

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

When the `CharacterData` is a `Text` [p.85] , or a `CDATASection` [p.90] , this attribute contains the property [character code] defined in [XML Information set]. When the `CharacterData` is a `Comment` [p.87] , this attribute contains the property [content] defined by the Comment Information Item in [XML Information set].

Exceptions on setting

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Exceptions on retrieval

`DOMException` [p.23]	DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` [p.17] variable on the implementation platform.

`length` of type `unsigned long`, readonly

The number of *16-bit units* [p.147] that are available through `data` and the `substringData` method below. This may have the value zero, i.e., `CharacterData` nodes may be empty.

Methods

`appendData`

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

Parameters

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

The `DOMString` to append.

Exceptions

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`deleteData`

Remove a range of *16-bit units* [p.147] 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.23]	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.147] offset.

Parameters

`offset` of type `unsigned long`

The character offset at which to insert.

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

The `DOMString` to insert.

Exceptions

`DOMException` [p.23]	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.147] offset with the specified string.

Parameters

`offset` of type `unsigned long`

The offset from which to start replacing.

`count` of type `unsigned long`

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

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

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

Exceptions

`DOMException` [p.23]	INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`substringData`

Extracts a range of data from the node.

Parameters

`offset` of type `unsigned long`

Start offset of substring to extract.

`count` of type `unsigned long`

The number of 16-bit units to extract.

Return Value

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

Exceptions

`DOMException` [p.23]	INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. DOMSTRING_SIZE_ERR: Raised if the specified range of text does not fit into a `DOMString` [p.17] .

Interface *Attr*

The `Attr` interface represents an attribute in an `Element` [p.77] object. Typically the allowable values for the attribute are defined in a document type definition.

`Attr` objects inherit the `Node` [p.49] 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.28] . However, they can be associated with `Element` [p.77] nodes contained within a `DocumentFragment`. In short, users and implementors of the DOM need to be aware that `Attr` nodes have some things in common with other objects inheriting the `Node` interface, but they also are quite distinct.

The attribute's effective value is determined as follows: if this attribute has been explicitly assigned any value, that value is the attribute's effective value; otherwise, if there is a declaration for this attribute, and that declaration includes a default value, then that default value is the attribute's effective value; otherwise, the attribute does not exist on this element in the structure model until it has been explicitly added. Note that the `nodeValue` attribute on the `Attr` instance can also be used to retrieve the string version of the attribute's value(s).

In XML, where the value of an attribute can contain entity references, the child nodes of the `Attr` node may be either `Text` [p.85] or `EntityReference` [p.95] nodes (when these are in use; see the description of `EntityReference` for discussion). Because the DOM Core is not aware of attribute types, it treats all attribute values as simple strings, even if the DTD or schema declares them as having *tokenized* [p.150] types.

The DOM implementation does not perform any *attribute value normalization*. While it is expected that the `value` and `nodeValue` attributes of an `Attr` node initially return the normalized value, this may not be the case after mutation. This is true, 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 entity references are involved, given that they are not represented in the DOM and they impact attribute value normalization.

Note: The properties [attribute type] and [references] defined in [XML Information set] are not accessible from DOM Level 3 Core. However, [DOM Level 3 Abstract Schemas and Load and Save] does provide a way to access the property [attribute type].

IDL Definition

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

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

Attributes

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

`ownerElement` of type `Element` [p.77] , readonly, introduced in **DOM Level 2**
   The `Element` [p.77] node this attribute is attached to or `null` if this attribute is not in use.
   This attribute represents the property [owner element] defined in [XML Information set].

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

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

   This attribute represents the property [specified] defined [XML Information set].

`value` of type `DOMString` [p.17]
   On retrieval, the value of the attribute is returned as a string. Character and general entity references are replaced with their values. See also the method `getAttribute` on the `Element` [p.77] interface.
   On setting, this creates a `Text` [p.85] node with the unparsed contents of the string. I.e. any characters that an XML processor would recognize as markup are instead treated as literal text. See also the method `setAttribute` on the `Element` [p.77] interface.
   If the `value` does contain the normalized attribute value, this attribute represents the property [normalized value] defined in [XML Information set].
   Exceptions on setting

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

Interface *Element*

The `Element` interface represents an *element* [p.148] in an HTML or XML document. Elements may have attributes associated with them; since the `Element` interface inherits from `Node` [p.49] , 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.75] 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.147] .

Note: In DOM Level 2, the method `normalize` is inherited from the `Node` [p.49] interface where it was moved.

Note: The properties [namespace attributes] and [in-scope namespaces] defined in [XML Information set] are not accessible from DOM Level 3 Core. However, [DOM Level 3 XPath] does provide a way to access the property [in-scope namespaces].

IDL Definition

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

Attributes

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

The name of the element. For example, in:

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

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

Methods

`getAttribute`

Retrieves an attribute value by name.

Parameters

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

The name of the attribute to retrieve.

Return Value

`DOMString` [p.17]	The `Attr` [p.75] 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.

Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.

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.17]

The *namespace URI* [p.149] of the attribute to retrieve.

`localName` of type `DOMString`

The *local name* [p.149] of the attribute to retrieve.

Return Value

`DOMString` [p.17]	The `Attr` [p.75] value as a string, or the empty string if that attribute does not have a specified or default value.

No Exceptions

`getAttributeNode`

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

Parameters

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

Return Value

`Attr` [p.75]	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.75] node by local name and namespace URI.
Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.
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.17]
The *namespace URI* [p.149] of the attribute to retrieve.
`localName` of type `DOMString`
The *local name* [p.149] of the attribute to retrieve.

Return Value

`Attr` [p.75]	The `Attr` node with the specified attribute local name and namespace URI or `null` if there is no such attribute.

No Exceptions

`getElementsByTagName`

Returns a `NodeList` [p.66] of all *descendant* [p.147] `Elements` with a given tag name, in *document order* [p.148] .

Parameters

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

Return Value

`NodeList` [p.66]	A list of matching `Element` nodes.

No Exceptions

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

Returns a `NodeList` [p.66] of all the *descendant* [p.147] `Elements` with a given local name and namespace URI in *document order* [p.148] .
Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.

Parameters

`namespaceURI` of type `DOMString` [p.17]
The *namespace URI* [p.149] of the elements to match on. The special value "*" matches all namespaces.

`localName` of type `DOMString`
The *local name* [p.149] of the elements to match on. The special value "*" matches all local names.

Return Value

`NodeList` [p.66]	A new `NodeList` object containing all the matched `Elements`.

No Exceptions

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

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

Parameters

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

Return Value

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

No Exceptions

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

Returns `true` when an attribute with a given local name and namespace URI is specified on this element or has a default value, `false` otherwise.
Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.
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.17]
The *namespace URI* [p.149] of the attribute to look for.
`localName` of type `DOMString`
The *local name* [p.149] of the attribute to look for.

Return Value

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

No Exceptions

`removeAttribute`

Removes an attribute by name. If the removed attribute is known to have a default value, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable. If the attribute does not have a specified or default value, calling 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.17]
The name of the attribute to remove.

Exceptions

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`removeAttributeNS` introduced in **DOM Level 2**

Removes an attribute by local name and namespace URI. If the removed attribute has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix. If the attribute does not have a specified or default value, calling this method has no effect.
Documents which do not support the "XML" feature will permit only the DOM Level 1 calls for creating/setting elements and attributes. Hence, if you specify a non-null namespace URI, these DOMs will never find a matching node.
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.17]
The *namespace URI* [p.149] of the attribute to remove.
`localName` of type `DOMString`
The *local name* [p.149] of the attribute to remove.

Exceptions

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

`removeAttributeNode`

Removes the specified attribute node. If the removed `Attr` [p.75] has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix, when applicable.

Parameters

`oldAttr` of type `Attr` [p.75]

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

Return Value

`Attr` [p.75]	The `Attr` node that was removed.

Exceptions

`DOMException` [p.23]	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.75] node plus any `Text` [p.85] and `EntityReference` [p.95] nodes, build the appropriate subtree, and use `setAttributeNode` to assign it as the value of an attribute.

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

Parameters

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

The name of the attribute to create or alter.

`value` of type `DOMString`

Value to set in string form.

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.
	NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

No Return Value

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

Adds a new attribute. If an attribute with the same local name and namespace URI is already present on the element, its prefix is changed to be the prefix part of the `qualifiedName`, and its value is changed to be the `value` parameter. This value is a simple string; it is not parsed as it is being set. So any markup (such as syntax to be recognized as an entity reference) is treated as literal text, and needs to be appropriately escaped by the implementation when it is written out. In order to assign an attribute value that contains entity references, the user must create an `Attr` [p.75] node plus any `Text` [p.85] and `EntityReference` [p.95] 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.17]
: The *namespace URI* [p.149] of the attribute to create or alter.

`qualifiedName` of type `DOMString`
: The *qualified name* [p.149] of the attribute to create or alter.

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

Exceptions

`DOMException` [p.23]	INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character, per the XML 1.0 specification [XML 1.0]. 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", or if the `qualifiedName`, or its prefix, is "xmlns" and the `namespaceURI` is different from "http://www.w3.org/2000/xmlns/". NOT_SUPPORTED_ERR: Always thrown if the current document does not support the `"XML"` feature, since namespaces were defined by XML.

No Return Value

`setAttributeNode`

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

Parameters

`newAttr` of type `Attr` [p.75]

The `Attr` node to add to the attribute list.

Return Value

`Attr` [p.75]	If the `newAttr` attribute replaces an existing attribute, the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.23]	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.75] 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.
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.75]

The `Attr` node to add to the attribute list.

Return Value

`Attr` [p.75]	If the `newAttr` attribute replaces an existing attribute with the same *local name* [p.149] and *namespace URI* [p.149] , the replaced `Attr` node is returned, otherwise `null` is returned.

Exceptions

`DOMException` [p.23]	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.75] nodes to re-use them in other elements. NOT_SUPPORTED_ERR: Always thrown if the current document does not support the `"XML"` feature, since namespaces were defined by XML.

Interface *Text*

The `Text` interface inherits from `CharacterData` [p.71] and represents the textual content (termed *character data* in XML) of an `Element` [p.77] or `Attr` [p.75] . 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.148] (elements, comments, etc.) and `Text` nodes that form the list of children of the element.

When a document is first made available via the DOM, there is only one `Text` node for each block of text. Users may create adjacent `Text` nodes that represent the contents of a given element without any intervening markup, but should be aware that there is no way to represent the separations between these nodes in XML or HTML, so they will not (in general) persist between DOM editing sessions. The `normalize` method on `Node` [p.49] merges any such adjacent `Text` objects into a single node for each block of text.

IDL Definition

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

Attributes

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

Returns whether this text node contains whitespace in element content, often abusively called "ignorable whitespace".

Note: An implementation can only return `true` if, one way or another, it has access to the relevant information (e.g., the DTD or schema).

This attribute represents the property [element content whitespace] defined in [XML Information set].

`wholeText` of type `DOMString` [p.17] , readonly, introduced in **DOM Level 3**

Returns all text of `Text` nodes *logically-adjacent text nodes* [p.148] to this node, concatenated in document order.

Methods

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

Substitutes the a specified text for the text of the current node and all *logically-adjacent text nodes* [p.148] .
This method returns the node in the hierarchy which received the replacement text, which is null if the text was empty or is the current node if the current node is not read-only or otherwise is a new node of the same type as the current node inserted at the site of the replacement. All *logically-adjacent text nodes* [p.148] are removed including the current node unless it was the recipient of the replacement text.
Where the nodes to be removed are read-only descendants of an `EntityReference` [p.95] , 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.90] nodes, the `replaceWholeText` method must fail before performing any modification of the document, raising a `DOMException` [p.23] with the code `NO_MODIFICATION_ALLOWED_ERR` [p.24] .

Parameters

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

The content of the replacing `Text` node.

Return Value

`Text` [p.85]	The `Text` node created with the specified content.

Exceptions

`DOMException` [p.23]	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.149] . 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.149] 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.147] offset at which to split, starting from `0`.

Return Value

`Text` [p.85]	The new node, of the same type as this node.

Exceptions

`DOMException` [p.23]	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.71] and represents the content of a comment, i.e., all the characters between the starting '`<!--`' and ending '`-->`'. Note that this is the definition of a comment in XML, and, in practice, HTML, although some HTML tools may implement the full SGML comment structure.

IDL Definition

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

Interface *UserDataHandler*

When associating an object to a key on a node using `setUserData` the application can provide a handler that gets called when the node the object is associated to is being cloned or imported. 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

```
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;

  void               handle(in unsigned short operation,
                            in DOMString key,
                            in DOMObject 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_CLONED`
: The node is cloned.

`NODE_DELETED`
: The node is deleted.

`NODE_IMPORTED`
: The node is imported.

`NODE_RENAMED`
: The node is renamed.

Methods

`handle`

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

Parameters

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

`key` of type `DOMString` [p.17]
: Specifies the key for which this handler is being called.

`data` of type `DOMObject` [p.19]
: Specifies the data for which this handler is being called.

`src` of type `Node` [p.49]
: Specifies the node being cloned, 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*

`DOMError` is an interface that describes an error.

IDL Definition

```
interface DOMError {
  const unsigned short      SEVERITY_WARNING               = 0;
  const unsigned short      SEVERITY_ERROR                 = 1;
  const unsigned short      SEVERITY_FATAL_ERROR           = 2;
  readonly attribute unsigned short  severity;
  readonly attribute DOMString       message;
  readonly attribute Object          relatedException;
  readonly attribute DOMLocator      location;
};
```

Constant *SEVERITY_WARNING*

The severity of the error described by the `DOMError` is warning

Constant *SEVERITY_ERROR*

The severity of the error described by the `DOMError` is error

Constant *SEVERITY_FATAL_ERROR*

The severity of the error described by the `DOMError` is fatal error

Attributes

`location` of type `DOMLocator` [p.90] , readonly

The location of the error.

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

An implementation specific string describing the error that occured.

`relatedException` of type `Object`, readonly

The related platform dependent exception if any.

Issue Error-1:

exception is a reserved word, we need to rename it.

Resolution: Change to "relatedException". (F2F 26 Sep 2001)

`severity` of type `unsigned short`, readonly

The severity of the error, either `SEVERITY_WARNING`, `SEVERITY_ERROR`, or `SEVERITY_FATAL_ERROR`.

Interface *DOMErrorHandler*

`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).

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

Issue ErrorHandler-1:

How does one register an error handler in the core? Passed as an argument to super-duper-normalize or registered on the DOMImplementation?

IDL Definition

```
interface DOMErrorHandler {
  boolean            handleError(in DOMError error);
};
```

Methods

`handleError`

This method is called on the error handler when an error occures.

Parameters

`error` of type `DOMError` [p.88]

The error object that describes the error, this object may be reused by the DOM implementation across multiple calls to the handleEvent method.

Return Value

`boolean`	If the handleError method returns `true` the DOM implementation should continue as if the error didn't happen when possible, if the method returns `false` then the DOM implementation should stop the current processing when possible.

No Exceptions

Interface *DOMLocator*

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

IDL Definition

```
interface DOMLocator {
  readonly attribute long            lineNumber;
  readonly attribute long            columnNumber;
  readonly attribute long            offset;
  readonly attribute Node            errorNode;
  readonly attribute DOMString       uri;
};
```

Attributes

`columnNumber` of type `long`, readonly
: The column number where the error occured, or -1 if there is no column number available.

`errorNode` of type `Node` [p.49] , readonly
: The DOM Node where the error occured, or null if there is no Node available.

`lineNumber` of type `long`, readonly
: The line number where the error occured, or -1 if there is no line number available.

`offset` of type `long`, readonly
: The byte or character offset into the input source, if we're parsing a file or a byte stream then this will be the byte offset into that stream, but if a character media is parsed then the offset will be the character offset. The value is `-1` if there is no offset available.

`uri` of type `DOMString` [p.17] , readonly
: The URI where the error occured, or null if there is no URI available.

1.3. Extended Interfaces

The interfaces defined here form part of the DOM Core specification, but objects that expose these interfaces will never be encountered in a DOM implementation that deals only with HTML.

The interfaces found within this section are not mandatory. A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` [p.25] interface 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 [p.22] . Please refer to additional information about Conformance [p.12] 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 `DOMString` [p.17] attribute of the `Text` [p.85] node holds the text that is contained by the CDATA section. Note that this *may* contain characters that need to be escaped outside of CDATA sections and that, depending on the character encoding ("charset") chosen for serialization, it may be impossible to write out some characters as part of a CDATA section.

The `CDATASection` interface inherits from the `CharacterData` [p.71] interface through the `Text` [p.85] interface. Adjacent `CDATASection` nodes are not merged by use of the `normalize` method of the `Node` [p.49] 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.29] has a `doctype` attribute whose value is either `null` or a `DocumentType` object. The `DocumentType` interface in the DOM Core provides an interface to the list of entities that are defined for the document, and little else because the effect of namespaces and the various XML schema efforts on DTD representation are not clearly understood as of this writing.

The DOM Level 2 doesn’t support editing `DocumentType` nodes.

Note: The property [children] defined by the Document Type Declaration Information Item in [XML Information set] is not accessible from DOM Level 3 Core.

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.67] , readonly

A `NamedNodeMap` [p.67] 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.93] interface.
The DOM Level 2 does not support editing entities, therefore `entities` cannot be altered in any way.

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

The internal subset as a string, 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.17] , readonly

The name of DTD; i.e., the name immediately following the `DOCTYPE` keyword.

`notations` of type `NamedNodeMap` [p.67] , readonly

A `NamedNodeMap` [p.67] containing the notations declared in the DTD. Duplicates are discarded. Every node in this map also implements the `Notation` [p.93] interface.
The DOM Level 2 does not support editing notations, therefore `notations` cannot be altered in any way.
This attribute represents the property [notations] defined by the Document Information Item in [XML Information set].

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

The public identifier of the external subset.
This attribute represents the property [public identifier] defined by the Document Type

Declaration Information Item in [XML Information set].

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

The system identifier of the external subset. This may be an absolute URI or not.
This attribute represents the property [system identifier] defined by the Document Type Declaration Information Item in [XML Information set].

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.49] is set to the declared name of the notation.

The DOM Level 1 does not support editing `Notation` nodes; they are therefore *readonly* [p.149] .

A `Notation` node does not have any parent.

Issue Notation-1:
adds a namespaceURI for notations?
Resolution: No. 1- notations are attached to a `DocumentType` [p.91] . 2- what would be the key for notations in namednodemap?

IDL Definition

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

Attributes

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

The public identifier of this notation. If the public identifier was not specified, this is `null`.
This attribute represents the property [public identifier] defined by the Notation Information Item in [XML Information set].

`systemId` of type `DOMString` [p.17] , 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.
This attribute represents the property [system identifier] defined by the Notation Information Item in [XML Information set].

Interface *Entity*

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

The `nodeName` attribute that is inherited from `Node` [p.49] 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.95] nodes in the document tree.

XML does not mandate that a non-validating XML processor read and process entity declarations made in the external subset or declared in external parameter entities. This means that parsed entities declared in the external subset need not be expanded by some classes of applications, and that the replacement 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.

The DOM Level 2 does not support editing `Entity` nodes; if a user wants to make changes to the contents of an `Entity`, every related `EntityReference` [p.95] 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.147] are *readonly* [p.149] .

An `Entity` node does not have any parent.

Note: If the entity contains an unbound *namespace prefix* [p.149] , the `namespaceURI` of the corresponding node in the `Entity` node subtree is `null`. The same is true for `EntityReference` [p.95] nodes that refer to this entity, when they are created using the `createEntityReference` method of the `Document` [p.29] interface. The DOM Level 2 does not support any mechanism to resolve namespace prefixes.

Note: The properties [notation name] and [notation] defined in [XML Information set] are not accessible from DOM Level 3 Core. However, [DOM Level 3 Abstract Schemas and Load and Save] does provide a way to access them.

IDL Definition

```
interface Entity : Node {
  readonly attribute DOMString       publicId;
  readonly attribute DOMString       systemId;
  readonly attribute DOMString       notationName;
  // Introduced in DOM Level 3:
           attribute DOMString       actualEncoding;
  // Introduced in DOM Level 3:
           attribute DOMString       encoding;
  // Introduced in DOM Level 3:
           attribute DOMString       version;
};
```

Attributes

`actualEncoding` of type `DOMString` [p.17] , introduced in **DOM Level 3**
: An attribute specifying the actual encoding of this entity, when it is an external parsed entity. This is `null` otherwise.

`encoding` of type `DOMString` [p.17] , 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.

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

`publicId` of type `DOMString` [p.17] , readonly
: The public identifier associated with the entity if specified, and `null` otherwise.
This attribute represents the property [public identifier] defined by the Unparsed Entity Information Item in [XML Information set].

`systemId` of type `DOMString` [p.17] , readonly
: The system identifier associated with the entity if specified, and `null` otherwise. This may be an absolute URI or not.
This attribute represents the property [system identifier] defined by the Unparsed Entity Information Item in [XML Information set].

`version` of type `DOMString` [p.17] , 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` objects may be inserted into the structure model when an entity reference is in the source document, or when the user wishes to insert an entity reference. Note that character references and references to predefined entities are considered to be expanded by the HTML or XML processor so that characters are represented by their Unicode equivalent rather than by an entity reference. Moreover, the XML processor may completely expand references to entities while building the structure model, instead of providing `EntityReference` objects. If it does provide such objects, then for a given `EntityReference` node, it may be that there is no `Entity` [p.93] node representing the referenced entity. If such an `Entity` exists, then the subtree of the `EntityReference` node is in general a copy of the `Entity` node subtree. However, this may not be true when an entity contains an unbound *namespace prefix* [p.149] . In such a case, because the namespace prefix resolution depends on where the entity reference is, the *descendants* [p.147] of the `EntityReference` node may be bound to different *namespace URIs* [p.149] .

As for `Entity` [p.93] nodes, `EntityReference` nodes and all their *descendants* [p.147] are *readonly* [p.149] .

Note: The properties [system identifier] and [public identifier] defined by the Unexpanded Entity Reference Information Item in [XML Information set] are accessible through the `Entity` [p.93] interface. The property [all declarations processed] is not accessible through the DOM API.

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.

Note: The property [notation] defined in [XML Information set] is not accessible from DOM Level 3 Core.

IDL Definition

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

};
```

Attributes

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

The content of this processing instruction. This is from the first non white space character after the target to the character immediately preceding the `?>`.
This attribute represents the property [content] defined by the Processing Instruction Information Item in [XML Information set].

Exceptions on setting

`DOMException` [p.23]	NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

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

The target of this processing instruction. XML defines this as being the first *token* [p.149] following the markup that begins the processing instruction.
This attribute represents the property [target] defined in [XML Information set].

Appendix A: Changes

Editors:
Arnaud Le Hors, IBM
Philippe Le Hégaret, W3C

A.1: Changes between DOM Level 2 Core and DOM Level 3 Core

To be completed...

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

OMG IDL
The DOM Level 2 specifications are now using Corba 2.3.1 instead of Corba 2.2.

Type `DOMString` [p.17]
The definition of `DOMString` [p.17] in IDL is now a `valuetype`.

A.2.1: Changes to DOM Level 1 Core interfaces and exceptions

Interface `Attr` [p.75]
The `Attr` [p.75] interface has one new attribute: `ownerElement`.

Interface `Document` [p.29]
The `Document` [p.29] interface has five new methods: `importNode`, `createElementNS`, `createAttributeNS`, `getElementsByTagNameNS` and `getElementById`.

Interface `NamedNodeMap` [p.67]
The `NamedNodeMap` [p.67] interface has three new methods: `getNamedItemNS`, `setNamedItemNS`, `removeNamedItemNS`.

Interface `Node` [p.49]
The `Node` [p.49] interface has two new methods: `isSupported` and `hasAttributes`.
`normalize`, previously in the `Element` [p.77] interface, has been moved in the `Node` [p.49] interface.
The `Node` [p.49] interface has three new attributes: `namespaceURI`, `prefix` and `localName`.
The `ownerDocument` attribute was specified to be `null` when the node is a `Document` [p.29] . It now is also `null` when the node is a `DocumentType` [p.91] which is not used with any `Document` yet.

Interface `DocumentType` [p.91]
The `DocumentType` [p.91] interface has three attributes: `publicId`, `systemId` and `internalSubset`.

Interface `DOMImplementation` [p.25]
The `DOMImplementation` [p.25] interface has two new methods: `createDocumentType` and `createDocument`.

Interface `Element` [p.77]

The `Element` [p.77] interface has eight new methods: `getAttributeNS`, `setAttributeNS`, `removeAttributeNS`, `getAttributeNodeNS`, `setAttributeNodeNS`, `getElementsByTagNameNS`, `hasAttribute` and `hasAttributeNS`.

The method `normalize` is now inherited from the `Node` [p.49] interface where it was moved.

Exception `DOMException` [p.23]

The `DOMException` [p.23] has five new exception codes: `INVALID_STATE_ERR` [p.24] , `SYNTAX_ERR` [p.24] , `INVALID_MODIFICATION_ERR` [p.24] , `NAMESPACE_ERR` [p.24] and `INVALID_ACCESS_ERR` [p.24] .

A.2.2: New features

A.2.2.1: New types

`DOMTimeStamp` [p.18]

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

Appendix B: Namespaces Algorithms

Editor:
Arnaud Le Hors, IBM

B.1: Namespace normalization

Namespace declaration attributes and prefixes are normalized as part of the `normalizeDocument` method of the `Document` [p.29] interface as if the following method described in pseudo code was called on the document element.

```
void Element.normalizeNamespaces()
{
  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 example 1
    }
    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 example 2
          // 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 has a colon in its name )
    {
       if ( Level 2 node )
       {
          ==> report an error
       }
       else
       {
          // Level 1 node

          if ( Name is not a QName )
          {
             ==> report an error
          }
          else
          {
             if ( Prefix is bound to something )
             {
                ==> report a warning
             }
             else
             {
                ==> report an error
             }
          }
       }
    }
    else
    {
      // Element has no namespace URI
      // Element has no pseudo-prefix
      if ( default Namespace in scope is "no namespace" )
      {
        ==> do nothing, we’re fine as we stand
      }
      else
      {
        if ( there’s a conflicting local default namespace declaration
             already present )
        {
          ==> change its value to use this empty namespace.
          See example 3
        }
        else
        {
          ==> Set the default namespace to "no namespace" by creating or
          changing a local declaration attribute: xmlns="".
          See example 4
        }
        // 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 ATTRS ////////
  for ( all Attrs of Element )
  {
    if ( Attr[i] has a namespace URI )
    {
      if ( Attr has no prefix, or has a prefix that conflicts with
           a binding already active in scope )
      {
        if ( Element is in the scope of a non default binding for this
             namespace )
        {
          if ( one or more prefix bindings are available )
          {
            if ( one is locally defined )
            {
              ==> pick that one.
            }
            else
            {
              ==> pick one arbitrarily
            }
            ==> Change the Attr to use that prefix.
          }
          else
          {
            ==> Create a local namespace declaration attr for this namespace
            with a prefix not already used in the current scope and following
            the pattern "NS" + index (starting at 1).
            Change the Attr to use this prefix.

            // 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
      {
        // prefix does match but....

        if ( namespace is "http://www.w3.org/2000/xmlns/" AND attribute does
             not have the prefix "xmlns:" or the nodeName "xmlns" )
        {
          // While all Namespace Declarations belong to a reserved NSURI,
          // it is _not_ true that all attributes having that NSURI are to be
          // considered Namespace Declarations.
          // According to the namespace spec, only "xmlns" and names having
          // the xmlns: prefix should be interpreted as declarations. So:
          if ( there is a non default binding for this namespace in scope
               with a prefix other than "xmlns" )
          {
            if ( one is locally defined )
            {
              ==> pick that one.
            }
            else
            {
              ==> pick one arbitrarily
            }
            ==> Change the Attr to use that prefix.
          }
          else
          {
            ==> Create a local namespace declaration attr for this namespace
            with a prefix not already used in the current scope and following
            the pattern "NS" + index (starting at 1).
            Change the Attr to use this prefix.

            // NOTE that this may break other nodes within thisElement’s
            // subtree, if they’re already using this prefix.
            // They will be repaired when we reach them.
          }
          // end non-namespace-decl with namespace-decl URI
        }
      }
```

```
      // end namespaced Attr
    }
    else
    {
      // Attr[i] has no namespace URI
      if ( Attr[i] has a colon in its name )
      {
         if ( Level 2 node )
         {
            ==> report an error
         }
         else
         {
            // Level 1 node
            if ( Name is not a QName )
            {
               ==> report an error
            }
            else
            {
               if ( Prefix is bound to something )
               {
                  ==> report a warning
               }
               else
               {
                  ==> report an error
               }
            }
         }
      }
      else
      {
        // attr has no namespace URI and no prefix
        // we're fine as we stand, 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.2: Namespace Prefix Lookup

The following describes in pseudo code the algorithm used in the `lookupNamespacePrefix` method of the `Node` [p.49] interface.

```
DOMString Element.lookupNamespacePrefix(in DOMString specifiedNamespaceURI)
{
   if ( Element’s namespaceURI == specifiedNamespaceURI )
   {
        return Element’s prefix
   }
   else if ( Element has an Attr and
             Attr’s namespaceURI == "http://www.w3.org/2000/xmlns/" and
             Attr’s prefix == "xmlns" and
             Attr’s value == specifiedNamespaceURI )
   {
        return Attr’s localName.
   }
   else if ( Element has an ancestor Element )
            // EntityReferences may have to be skipped to get to it
   {
        return ancestorElement.lookupNamespacePrefix(specifiedNamespaceURI)
   }
   else {
        return unknown (null)
   }
}
```

Issue lookupNamespacePrefixAlgo-1:
    Isn’t the name the opposite of what it stands for?
Issue lookupNamespacePrefixAlgo-2:
    How does one differentiate the case where it’s the default namespace (prefix == null) from the case where the namespaceURI was not found?
Issue lookupNamespacePrefixAlgo-3:
    How does one specify this is for an attribute and therefore the default namespace is not applicable?

B.3: Namespace URI Lookup

The following describes in pseudo code the algorithm used in the `lookupNamespaceURI` method of the `Node` [p.49] interface.

```
DOMString Element.lookupNamespaceURI(in DOMString specifiedPrefix)
{
   return lookupNamespaceURI(specifiedPrefix, this);
}

DOMString Element.lookupNamespaceURI(in DOMString specifiedPrefix, Element el)
{
   if ( Element’s namespace URI != null and
        Element’s prefix == specifiedPrefix and
        el.lookupNamespacePrefix(Element’s namespace URI) == specifiedPrefix )
   {
        return Element’s namespace URI
   }
   else if ( Element has an Attr and
             Attr’s namespaceURI == "http://www.w3.org/2000/xmlns/" and
             Attr’s prefix == "xmlns" and
             Attr’s localName == specifiedPrefix and
         el.lookupNamespacePrefix(Attr’s value URI) == specifiedPrefix )
   {
        return Attr’s value.
   }
   else if ( Element has an ancestor Element )
            // EntityReferences may have to be skipped to get to it
   {
        return ancestorElement.lookupNamespaceURI(specifiedPrefix, el)
   }
   else {
        return unknown (null)
   }
}
```

Issue lookupNamespaceURIAlgo-1:

How does one look for the default namespace?

Appendix C: Accessing code point boundaries

Mark Davis, IBM
Lauren Wood, SoftQuad Software Inc.

C.1: Introduction

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

Characters are represented in Unicode by numbers called *code points* (also called *scalar values*). These numbers can range from 0 up to 1,114,111 = $10FFFF_{16}$ (although some of these values are illegal). Each code point can be directly encoded with a 32-bit code unit. This encoding is termed UCS-4 (or UTF-32). The DOM specification, however, uses UTF-16, in which the most frequent characters (which have values less than $FFFF_{16}$) are represented by a single 16-bit code unit, while characters above $FFFF_{16}$ use a special pair of code units called a *surrogate pair*. For more information, see [Unicode 3.0] or the Unicode Web site.

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

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

C.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 D: IDL Definitions

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

The IDL files are also available as:
http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020114/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   Object DOMKeyObject;

  typedef   Object DOMObject;

  interface DOMImplementation;
  interface DocumentType;
  interface Document;
  interface NodeList;
  interface NamedNodeMap;
  interface UserDataHandler;
  interface Element;
  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;


interface DOMImplementationSource {
  DOMImplementation  getDOMImplementation(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:
  DOMImplementation  getInterface(in DOMString feature);
};

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

// Introduced in DOM Level 2:
readonly attribute DOMString       localName;
// Introduced in DOM Level 2:
boolean            hasAttributes();
// Introduced in DOM Level 3:
readonly attribute DOMString       baseURI;

// TreePosition
const unsigned short      TREE_POSITION_PRECEDING        = 0x01;
const unsigned short      TREE_POSITION_FOLLOWING        = 0x02;
const unsigned short      TREE_POSITION_ANCESTOR         = 0x04;
const unsigned short      TREE_POSITION_DESCENDANT       = 0x08;
const unsigned short      TREE_POSITION_EQUIVALENT       = 0x10;
const unsigned short      TREE_POSITION_SAME_NODE        = 0x20;
const unsigned short      TREE_POSITION_DISCONNECTED     = 0x00;

// Introduced in DOM Level 3:
unsigned short     compareTreePosition(in Node other);
// 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          lookupNamespacePrefix(in DOMString namespaceURI);
    // Introduced in DOM Level 3:
    DOMString          lookupNamespaceURI(in DOMString prefix);
    // Introduced in DOM Level 3:
    boolean            isEqualNode(in Node arg, 
                                   in boolean deep);
    // Introduced in DOM Level 3:
    Node               getInterface(in DOMString feature);
    // Introduced in DOM Level 3:
    DOMKeyObject       setUserData(in DOMString key, 
                                   in DOMKeyObject data, 
                                   in UserDataHandler handler);
    // Introduced in DOM Level 3:
    DOMKeyObject       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);
    // Introduced in DOM Level 2:
    Node               setNamedItemNS(in Node arg)
                                        raises(DOMException);
    // Introduced in DOM Level 2:
    Node               removeNamedItemNS(in DOMString namespaceURI, 
                                         in DOMString localName)
                                        raises(DOMException);
  };

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

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

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

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

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

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

```
};

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

interface Comment : CharacterData {
};

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;

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

interface DOMError {
  const unsigned short      SEVERITY_WARNING               = 0;
  const unsigned short      SEVERITY_ERROR                 = 1;
  const unsigned short      SEVERITY_FATAL_ERROR           = 2;
  readonly attribute unsigned short  severity;
  readonly attribute DOMString       message;
  readonly attribute Object          relatedException;
  readonly attribute DOMLocator      location;
};

interface DOMErrorHandler {
  boolean            handleError(in DOMError error);
};

interface DOMLocator {
  readonly attribute long            lineNumber;
  readonly attribute long            columnNumber;
  readonly attribute long            offset;
  readonly attribute Node            errorNode;
  readonly attribute DOMString       uri;
};

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:
           attribute DOMString       actualEncoding;
  // Introduced in DOM Level 3:
           attribute DOMString       encoding;
  // Introduced in DOM Level 3:
           attribute DOMString       version;
};

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:
         attribute DOMString       actualEncoding;
// Introduced in DOM Level 3:
         attribute DOMString       encoding;
// Introduced in DOM Level 3:
         attribute boolean         standalone;
// Introduced in DOM Level 3:
         attribute DOMString       version;
                                 // raises(DOMException) on setting

// Introduced in DOM Level 3:
         attribute boolean         strictErrorChecking;
// Introduced in DOM Level 3:
         attribute DOMErrorHandler errorHandler;
// 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:
void                normalizeDocument();
// Introduced in DOM Level 3:
boolean             canSetNormalizationFeature(in DOMString name,
                                           in boolean state);
// Introduced in DOM Level 3:
void                setNormalizationFeature(in DOMString name,
                                        in boolean state)
                                 raises(DOMException);
// Introduced in DOM Level 3:
boolean             getNormalizationFeature(in DOMString name)
                                 raises(DOMException);
// Introduced in DOM Level 3:
Node                renameNode(in Node n,
                           in DOMString namespaceURI,
                           in DOMString name)
```

```
                                        raises(DOMException);
  };
};

#endif // _DOM_IDL_
```

dom.idl:

Appendix E: 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/2002/WD-DOM-Level-3-Core-20020114/java-binding.zip

E.1: Java Binding Extension

This section defines the `DOMImplementationRegistry` object, discussed in Bootstrapping [p.22] , 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.25] interface.

org/w3c/dom/DOMImplementationRegistry.java:

```
package org.w3c.dom;

import java.util.StringTokenizer;
import java.util.Vector;

/**
 * This class holds the list of registered DOMImplementations. It is first
 * initialized based on the content of the space separated list of classnames
 * contained in the System Property "org.w3c.dom.DOMImplementationSourceList".
 *
 * <p>Subsequently, additional sources can be registered and implementations
 * can be queried based on a list of requested features.
 *
 * <p>This provides an application with an implementation independent starting
 * point.
 *
 * @see DOMImplementation
 * @see DOMImplementationSource
 */
public class DOMImplementationRegistry
{

    // The system property to specify the DOMImplementationSource class names.
    public static String PROPERTY = "org.w3c.dom.DOMImplementationSourceList";

    private static Vector sources = new Vector();
    private static boolean initialized = false;

    private static void initialize() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException
    {
        initialized = true;
        String p = System.getProperty(PROPERTY);
    if (p == null) {
        return;
    }
    StringTokenizer st = new StringTokenizer(p);
    while (st.hasMoreTokens()) {
        Object source = Class.forName(st.nextToken()).newInstance();
        sources.addElement(source);
    }
}

/**
 * Return the first registered implementation that has the desired features,
 * or null 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 this source has none.
 */
public static DOMImplementation getDOMImplementation(String features)
    throws ClassNotFoundException,
    InstantiationException, IllegalAccessException
{
    if (!initialized) {
        initialize();
    }
    int len = sources.size();
    for (int i = 0; i < len; i++) {
        DOMImplementationSource source =
            (DOMImplementationSource) sources.elementAt(i);

        DOMImplementation impl = source.getDOMImplementation(features);
        if (impl != null) {
            return impl;
        }
    }
    return null;
}

/**
 * Register an implementation.
 */
public static void addSource(DOMImplementationSource s)
    throws ClassNotFoundException,
    InstantiationException, IllegalAccessException
{
    if (!initialized) {
        initialize();
    }
    sources.addElement(s);
    // update system property accordingly
    StringBuffer b = new StringBuffer(System.getProperty(PROPERTY));
```

```
        b.append(" " + s.getClass().getName());
        System.setProperty(PROPERTY, b.toString());
    }
}
```

With this, the first line of an application typically becomes something like (modulo exception handling):

```
    DOMImplementation impl = DOMImplementationRegistry.getDOMImplementation("XML 1.0");
```

Issue Level-3-Java-Bootstrap-1:
Should this provides for handling more than one implementation at a time?
Resolution: Yes.

Issue Level-3-Java-Bootstrap-2:
Should this be even simpler and force the implementation to provide this class (and not necessarily rely on any system property)?
Resolution: No.

Issue Level-3-Java-Bootstrap-3:
This requires all DOMImplementationSources to be pre-instantiated.
Resolution: Proposed: It’s ok.

Issue Level-3-Java-Bootstrap-4:
Some people may like to be able to enumerate available implementations. DOMImplementation objects may be too dynamic to enumerate. We should explore any significant use case that cannot be solved by this proposal.
Resolution: No real need. Additional features can be used to further differentiate implementations.

Issue Level-3-Java-Bootstrap-5:
A space-separated feature string may not be the optimal way to pass a feature list. It was motivated by the lack of an array construct.
Resolution: Proposed: It’s ok.

Issue Level-3-Java-Bootstrap-6:
Should "*" given as the version number be interpreted as "any version". hasFeature() does not allow this, it requires a specific version to be given.
Resolution: No. (telcon xxxx)

E.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;

}
```

org/w3c/dom/DOMImplementationSource.java:

```
package org.w3c.dom;

public interface DOMImplementationSource {
    public DOMImplementation getDOMImplementation(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 DOMImplementation getInterface(String feature);

}
```

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 getActualEncoding();
    public void setActualEncoding(String actualEncoding);

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

    public boolean getStandalone();
    public void setStandalone(boolean standalone);

    public String getVersion();
    public void setVersion(String version)
                              throws DOMException;

    public boolean getStrictErrorChecking();
    public void setStrictErrorChecking(boolean strictErrorChecking);

    public DOMErrorHandler getErrorHandler();
    public void setErrorHandler(DOMErrorHandler errorHandler);

    public String getDocumentURI();
    public void setDocumentURI(String documentURI);

    public Node adoptNode(Node source)
                      throws DOMException;

    public void normalizeDocument();

    public boolean canSetNormalizationFeature(String name,
                                              boolean state);

    public void setNormalizationFeature(String name,
                                        boolean state)
                                        throws DOMException;

    public boolean getNormalizationFeature(String name)
                                           throws DOMException;

    public Node renameNode(Node n,
                       String namespaceURI,
                       String name)
                       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();

    // TreePosition
    public static final short TREE_POSITION_PRECEDING   = 0x01;
    public static final short TREE_POSITION_FOLLOWING   = 0x02;
    public static final short TREE_POSITION_ANCESTOR    = 0x04;
    public static final short TREE_POSITION_DESCENDANT  = 0x08;
    public static final short TREE_POSITION_EQUIVALENT  = 0x10;
    public static final short TREE_POSITION_SAME_NODE   = 0x20;
    public static final short TREE_POSITION_DISCONNECTED = 0x00;

    public short compareTreePosition(Node other);

    public String getTextContent()
                                  throws DOMException;
    public void setTextContent(String textContent)
                                  throws DOMException;

    public boolean isSameNode(Node other);

    public String lookupNamespacePrefix(String namespaceURI);

    public String lookupNamespaceURI(String prefix);

    public boolean isEqualNode(Node arg,
                               boolean deep);

    public Node getInterface(String feature);

    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);

    public Node setNamedItemNS(Node arg)
                               throws DOMException;

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

}
```

org/w3c/dom/CharacterData.java:

```
package org.w3c.dom;

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

    public int getLength();

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

    public void appendData(String arg)
                           throws DOMException;

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

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

}
```

org/w3c/dom/Attr.java:

```
package org.w3c.dom;

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

    public boolean getSpecified();

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

    public Element getOwnerElement();

}
```

org/w3c/dom/Element.java:

```
package org.w3c.dom;

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

    public String getAttribute(String name);

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

    public void removeAttribute(String name)
                                throws DOMException;

    public Attr getAttributeNode(String name);

    public Attr setAttributeNode(Attr newAttr)
                                 throws DOMException;

    public Attr removeAttributeNode(Attr oldAttr)
                                    throws DOMException;

    public NodeList getElementsByTagName(String name);

    public String getAttributeNS(String namespaceURI,
                                 String localName);

    public void setAttributeNS(String namespaceURI,
                               String qualifiedName,
                               String value)
```

```
                                  throws DOMException;

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

    public Attr getAttributeNodeNS(String namespaceURI,
                                   String localName);

    public Attr setAttributeNodeNS(Attr newAttr)
                                   throws DOMException;

    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName);

    public boolean hasAttribute(String name);

    public boolean hasAttributeNS(String namespaceURI,
                                  String localName);

}
```

org/w3c/dom/Text.java:

```
package org.w3c.dom;

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

    public boolean getIsWhitespaceInElementContent();

    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/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 void handle(short operation,
                       String key,
                       Object data,
                       Node src,
                       Node dst);

}
```

org/w3c/dom/DOMError.java:

```
package org.w3c.dom;

public interface DOMError {
    public static final short SEVERITY_WARNING          = 0;
    public static final short SEVERITY_ERROR            = 1;
    public static final short SEVERITY_FATAL_ERROR      = 2;
    public short getSeverity();

    public String getMessage();

    public Object getRelatedException();

    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 getOffset();

    public Node getErrorNode();

    public String getUri();

}
```

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 getActualEncoding();
    public void setActualEncoding(String actualEncoding);

    public String getEncoding();
    public void setEncoding(String encoding);
    public String getVersion();
    public void setVersion(String version);

}
```

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 F: ECMAScript Language Binding

This appendix contains the complete ECMAScript [ECMAScript] binding for the Level 3 Document Object Model Core definitions.

F.1: ECMAScript Binding Extension

This section defines the `DOMImplementationRegistry` object, discussed in Bootstrapping [p.22] , 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**.

sources

This property is an **Array**. It contains all registered objects that implement the **DOMImplementationSource** interface.

F.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**.

Objects that implement the **DOMException** interface:
Properties of objects that implement the **DOMException** interface:
code
This property is a **Number**.

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**.

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.
getInterface(feature)
This function returns an object that implements the **DOMImplementation** interface.
The **feature** 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.

actualEncoding

This property is a **String**.

encoding

This property is a **String**.

standalone

This property is a **Boolean**.

version

This property is a **String** and can raise an objewct that implements **DOMException** interface on setting.

strictErrorChecking

This property is a **Boolean**.

errorHandler

This property is an object that implements the **DOMErrorHandler** interface.

documentURI

This property is a **String**.

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.

canSetNormalizationFeature(name, state)

This function returns a **Boolean**.
The **name** parameter is a **String**.
The **state** parameter is a **Boolean**.

setNormalizationFeature(name, state)

This function has no return value.

The **name** parameter is a **String**.
The **state** parameter is a **Boolean**.
This function can raise an object that implements the **DOMException** interface.

getNormalizationFeature(name)
This function returns a **Boolean**.
The **name** parameter is a **String**.
This function can raise an object that implements the **DOMException** interface.

renameNode(n, namespaceURI, name)
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 **name** 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.TREE_POSITION_PRECEDING
The value of the constant **Node.TREE_POSITION_PRECEDING** is **0x01**.

Node.TREE_POSITION_FOLLOWING
The value of the constant **Node.TREE_POSITION_FOLLOWING** is **0x02**.

Node.TREE_POSITION_ANCESTOR
The value of the constant **Node.TREE_POSITION_ANCESTOR** is **0x04**.

Node.TREE_POSITION_DESCENDANT
The value of the constant **Node.TREE_POSITION_DESCENDANT** is **0x08**.

Node.TREE_POSITION_EQUIVALENT
The value of the constant **Node.TREE_POSITION_EQUIVALENT** is **0x10**.
Node.TREE_POSITION_SAME_NODE
The value of the constant **Node.TREE_POSITION_SAME_NODE** is **0x20**.
Node.TREE_POSITION_DISCONNECTED
The value of the constant **Node.TREE_POSITION_DISCONNECTED** is **0x00**.

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 **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 objewct that implements **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 **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**.

compareTreePosition(other)
This function returns a **Number**.
The **other** parameter is an object that implements the **Node** interface.

isSameNode(other)
This function returns a **Boolean**.
The **other** parameter is an object that implements the **Node** interface.

lookupNamespacePrefix(namespaceURI)
This function returns a **String**.
The **namespaceURI** parameter is a **String**.

lookupNamespaceURI(prefix)
This function returns a **String**.
The **prefix** parameter is a **String**.

isEqualNode(arg, deep)
This function returns a **Boolean**.
The **arg** parameter is an object that implements the **Node** interface.
The **deep** parameter is a **Boolean**.

getInterface(feature)
This function returns an object that implements the **Node** interface.
The **feature** 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**.

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 **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 objewct that implements **DOMException** interface on setting.

ownerElement

This read-only property is an object that implements the **Element** interface.

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**.

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**.

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**.

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**.

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**.

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:

isWhitespaceInElementContent

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 **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**.

Objects that implement the **UserDataHandler** interface:

Functions of objects that implement the **UserDataHandler** interface:

handle(operation, key, data, src, dst)

This function has no return value.
The **operation** parameter is a **Number**.
The **key** parameter is a **String**.
The **data** parameter is an object that implements the **Object** interface.
The **src** parameter is an object that implements the **Node** interface.
The **dst** 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 **0**.

DOMError.SEVERITY_ERROR

The value of the constant **DOMError.SEVERITY_ERROR** is **1**.

DOMError.SEVERITY_FATAL_ERROR

The value of the constant **DOMError.SEVERITY_FATAL_ERROR** is **2**.

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**.

relatedException

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.

Objects that implement the **DOMErrorHandler** interface:

Functions of objects that implement the **DOMErrorHandler** interface:

handleError(error)
This function returns a **Boolean**.
The **error** 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**.
offset
This read-only property is a **Number**.
errorNode
This read-only property is an object that implements the **Node** interface.
uri
This read-only property is a **String**.

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**.

actualEncoding

This property is a **String**.

encoding

This property is a **String**.

version

This 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 objewct that implements **DOMException** interface on setting.

Appendix G: Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including members 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), 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 Chair*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL), 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!).

G.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)

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

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

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

API
An *API* is an Application Programming Interface, a set of functions or methods used to access some functionality.

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

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

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

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

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

descendant
A *descendant* node of any node A is any node below A in a tree model of a document, where "below" means "away from the root."

document element
There is only one document element in a `Document` [p.29] . 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.147] 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.85] or `CDataSection` nodes that may be visited sequentially in *document order* [p.148] without entering, exiting, or passing over `Element` [p.77] , `Comment` [p.87] , or `ProcessingInstruction` [p.95] 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 4.0]

inheritance

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

interface

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

language binding

A programming *language binding* for an IDL specification is an implementation of the interfaces in the specification for the given language. For example, a Java language binding for the Document

Object Model IDL specification would implement the concrete Java classes that provide the functionality exposed by the interfaces.

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

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.150] (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.

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 3.0].

token
An information item such as an XML Name which has been *tokenized* [p.150] .

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

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

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]

XML name
: See *XML name* in the XML specification ([XML 1.0]).

XML namespace
: An *XML namespace* is a collection of names, identified by a URI reference [RFC2396], which are used in XML documents as element types and attribute names. [XML Namespaces]

References

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

I.1: Normative references

CharModel
W3C (World Wide Web Consortium) Character Model for the World Wide Web, January 2001. Available at http://www.w3.org/TR/2001/WD-charmod-20010126

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

DOM Level 2 Core
W3C (World Wide Web Consortium) Document Object Model Level 2 Core Specification, November 2000. Available at http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113

DOM Level 2 HTML
W3C (World Wide Web Consortium) Document Object Model Level 2 HTML Specification, November 2000. Available at http://www.w3.org/TR/2000/WD-DOM-Level-2-HTML-20001113

ECMAScript
ISO (International Organization for Standardization). ISO/IEC 16262:1998. ECMAScript Language Specification. Available from ECMA (European Computer Manufacturers Association) at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

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

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

MathML 2.0
W3C (World Wide Web Consortium) Mathematical Markup Language (MathML) Version 2.0, February 2001. Available at http://www.w3.org/TR/MathML2

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

SVG 1.0
W3C (World Wide Web Consortium) Scalable Vector Graphics (SVG) 1.0 Specification, September 2001. Available at http://www.w3.org/TR/SVG

Unicode 3.0
The Unicode Consortium. The Unicode Standard, Version 3.0., 2000, Reading, Mass.: Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

XML 1.0
W3C (World Wide Web Consortium) Extensible Markup Language (XML) 1.0, October 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006

XML 1.1
W3C (World Wide Web Consortium) Extensible Markup Language (XML) 1.1, December 2001. Available at http://www.w3.org/TR/2001/WD-xml11-20011213/

XML Base
W3C (World Wide Web Consortium) XML Base, June 2001. Available at http://www.w3.org/TR/xmlbase

XML Information set
W3C (World Wide Web Consortium) XML Information Set, October 2001. Available at http://www.w3.org/TR/2001/REC-xml-infoset-20011024

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

I.2: Informative references

Canonical XML
W3C (World Wide Web Consortium) Canonical XML, March 2001. Available at http://www.w3.org/TR/2001/REC-xml-c14n-20010315

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

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

DOM Level 3 Abstract Schemas and Load and Save
W3C (World Wide Web Consortium) Document Object Model Level 3 Abstract Schemas and Load and Save Specification, June 2001. Available at http://www.w3.org/TR/DOM-Level-3-ASLS

DOM Level 3 Events
W3C (World Wide Web Consortium) Document Object Model Level 3 Events Specification, August 2001. Available at http://www.w3.org/TR/DOM-Level-3-Events

DOM Level 3 XPath
W3C (World Wide Web Consortium) Document Object Model Level 3 XPath Specification, August 2001. Available at http://www.w3.org/TR/DOM-Level-3-XPath

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

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

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

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

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

RFC2396

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

XPointer

W3C (World Wide Web Consortium) XML Pointer Language (XPointer), January 2001. Available at http://www.w3.org/TR/xptr

Index