Document Object Model (DOM) Level 3 Core Specification

1.0

W3C Recommendation 07 April 2004

This version:

http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407

Latest version:

http://www.w3.org/TR/DOM-Level-3-Core

Previous version:

http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205/

Authors and Contributors:

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

Abstract

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

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

Status of this document

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

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

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

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

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

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

Table of Contents

Preface

Appendices

This page is intentionally left blank.

A. Expanded Table of Contents

B. W3C Copyright Notices and Licenses

This document is published under the Preface B.1 – W3C® Document Copyright Notice and License on page 1. The bindings within this document are published under the Preface B.2 – W3C® Software Copyright Notice and License on page 2. The software license requires "Notice of any changes or modifications to the W3C files, including the date changes were made." Consequently, modified versions of the DOM bindings must document that they do not conform to the W3C standard; in the case of the IDL definitions, the pragma prefix can no longer be 'w3c.org'; in the case of the Java language binding, the package names can no longer be in the 'org.w3c' package.

B.1. W3C® Document Copyright Notice and License

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

B.2. W3C® Software Copyright Notice and License

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

B.3. W3C® Short Software Notice

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

C. What is the Document Object Model?

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

C.1. Introduction

The Document Object Model (DOM) is an application programming interface () for valid and well-formed 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 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 . The DOM is designed to be used with any programming language. In order to provide a precise, language-independent specification of the DOM interfaces, we have chosen to define the specifications in Object Management Group (OMG) IDL [OMG IDL], as defined in the CORBA 2.3.1 specification [CORBA]. In addition to the OMG IDL specification, we provide for Java [Java] and ECMAScript [ECMAScript] (an industry-standard scripting language based on JavaScript [JavaScript] and JScript [JScript]). Because of language binding restrictions, a mapping has to be applied between the OMG IDL and the programming language in used. For example, while the DOM uses IDL attributes in the definition of interfaces, Java does not allow interfaces to contain attributes:

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

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

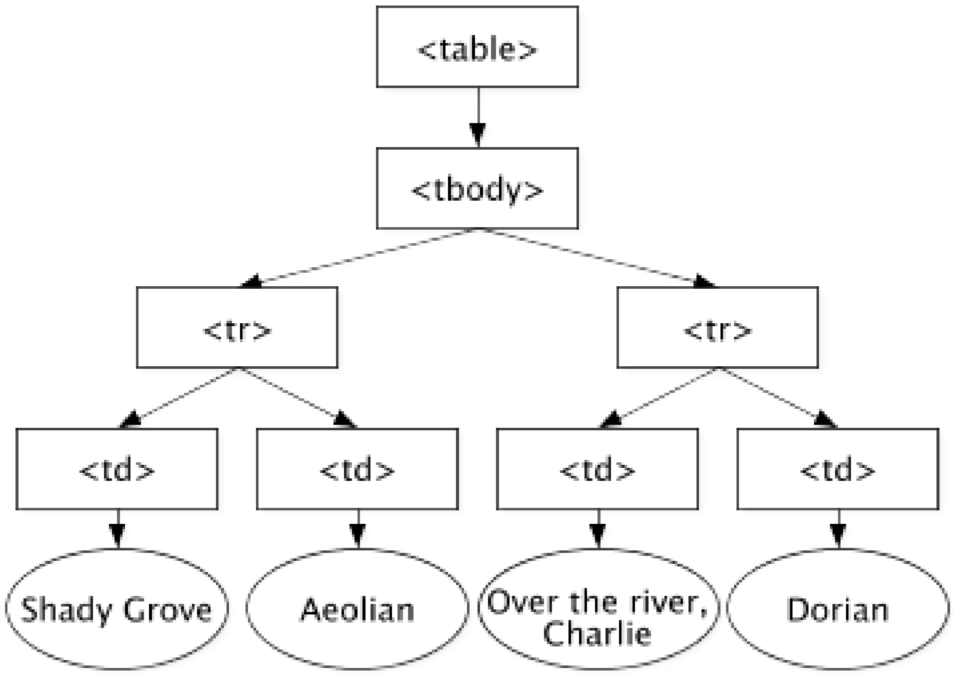
☞ OMG IDL is used only as a language-independent and implementation-neutral way to specify . Various other IDLs could have been used ([COM], [Java IDL], [MIDL], ...). In general, IDLs are designed for specific computing environments. The Document Object Model can be implemented in any computing environment,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

C.2. What the Document Object Model is

The DOM is a programming for documents. It is based on an object structure that closely resembles the structure of the documents it . For instance, consider this table, taken from an XHTML document:

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

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



An example of DOM manipulation using ECMAScript would be:

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

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

// remove its first td element
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

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

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

☞ 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 "" 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 , not by an object model. In an abstract , 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.

C.3. 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 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 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.

C.4. 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.

C.5. 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 § 1 – Document Object Model Core on page 10.

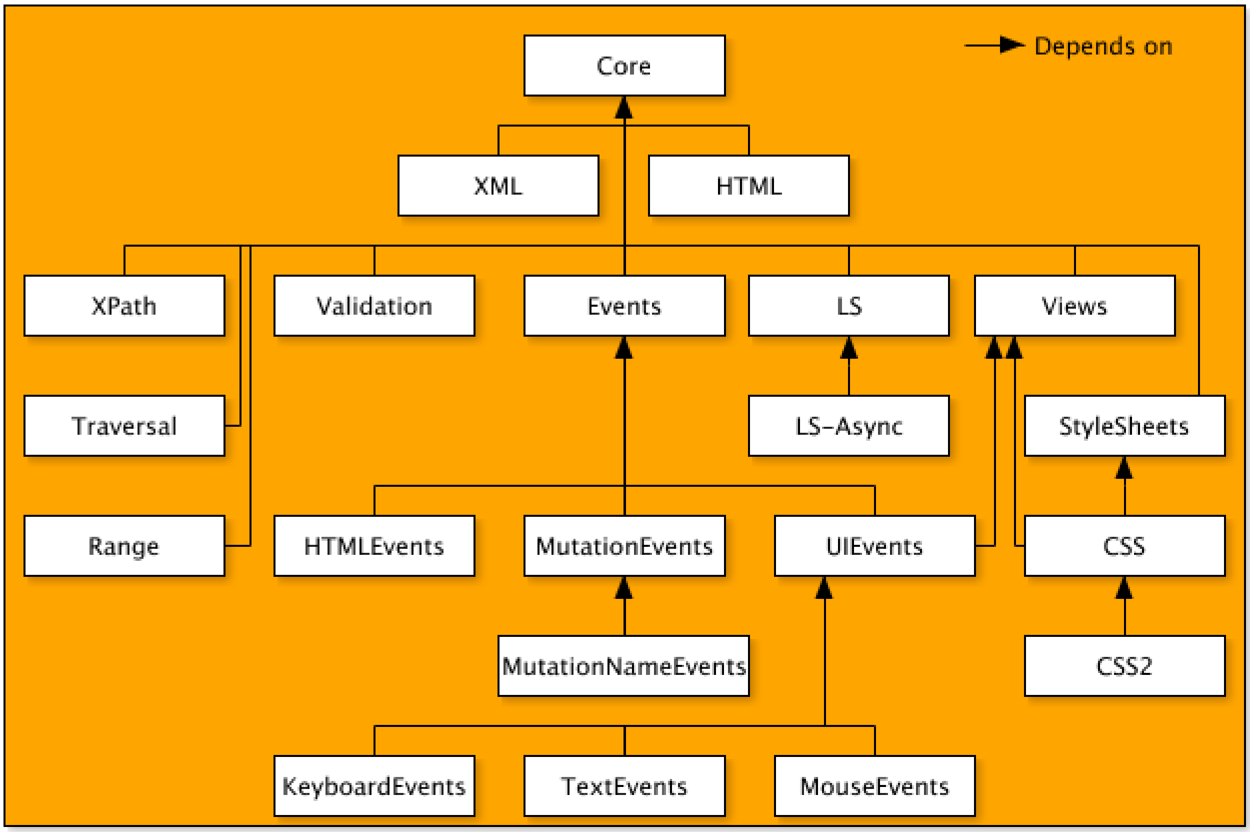
Note: When a DOM representation of a document is serialized as XML or HTML text, applications will need to check each character in text data to see if it needs to be escaped using a numeric or pre-defined entity. Failing to do so could result in invalid HTML or XML. Also, should be aware of the fact 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.

C.6. DOM Architecture

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

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

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



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

C.7. Conformance

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

An implementation is DOM Level 3 conformant if it supports the Core module defined in this document (see § 1.4 – Fundamental Interfaces: Core Module on page 18). 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".

XML module

Defines the feature "XML".

Events module

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

User interface Events module

defines the feature "UIEvents" in [DOM Level 3 Events].

Mouse Events module

defines the feature "MouseEvents" in [DOM Level 3 Events].

Text Events module

defines the feature "TextEvents" in [DOM Level 3 Events].

Keyboard Events module

defines the feature "KeyboardEvents" in [DOM Level 3 Events].

Mutation Events module

defines the feature "MutationEvents" in [DOM Level 3 Events].

Mutation name Events module

defines the feature "MutationNameEvents" in [DOM Level 3 Events].

HTML Events module

defines the feature "HTMLEvents" in [DOM Level 3 Events].

Load and Save module

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

Asynchronous load module

defines the feature "LS-Async" in [DOM Level 3 Load and Save].

Validation module

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

XPath module

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

A DOM implementation must not return `true` to the `DOMImplementation.hasFeature(feature, version)` of the `DOMImplementation` 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"`.

C.8. 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

Arnaud Le Hors, IBM; Philippe Le Hégaret, W3C; Gavin Nicol, Inso EPS (for DOM Level 1); Lauren Wood, SoftQuad, Inc. (for DOM Level 1); Mike Champion, Arbortext and Software AG (for DOM Level 1 from November 20, 1997); Steve Byrne, JavaSoft (for DOM Level 1 until November 19, 1997)
This specification defines a set of objects and interfaces for accessing and manipulating document objects. The functionality specified (the *Core* functionality) is sufficient to allow software developers and Web script authors to access and manipulate parsed HTML [HTML 4.01] and XML [XML 1.0] content inside conforming products. The DOM Core also allows creation and population of a `Document` object using only DOM API calls. A solution for loading a `Document` and saving it persistently is proposed in [DOM Level 3 Load and Save].

1.1. Overview of the DOM Core Interfaces

1.1.1. The DOM Structure Model

The DOM presents documents as a hierarchy of `Node` objects that also implement other, more specialized interfaces. Some types of nodes may have 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` -- `Element` (maximum of one), `ProcessingInstruction`, `Comment`, `DocumentType` (maximum of one)
- `DocumentFragment` -- `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- `DocumentType` -- no children
- `EntityReference` -- `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- `Element` -- `Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDATASection`, `EntityReference`
- `Attr` -- `Text`, `EntityReference`
- `ProcessingInstruction` -- no children

- `Comment` -- no children
- `Text` -- no children
- `CDATASection` -- no children
- `Entity` -- `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- `Notation` -- no children

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

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

1.1.3. Naming Conventions

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

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

1.1.4. Inheritance vs. Flattened Views of the API

The DOM Core present two somewhat different sets of interfaces to an XML/HTML document: one presenting an "object oriented" approach with a hierarchy of , and a "simplified" view that allows all manipulation to be done via the `Node` interface without requiring casts (in Java and other C-like languages) or query interface calls in 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 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 .

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

1.2. Basic Types

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

1.2.1. The `DOMString` Type

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

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

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

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

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` (a high surrogate and a low surrogate). For issues related to string comparisons, refer to § 1.3.1 – String Comparisons in the DOM on page 13.

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

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

1.2.2. The `DOMTimeStamp` Type

The `DOMTimeStamp` type is used to store an absolute or relative time.

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

1.2.3. The `DOMUserData` Type

The `DOMUserData` type is used to store application data.

For Java, `DOMUserData` is bound to the `Object` type. For ECMAScript, `DOMUserData` is bound to `any type`.

1.2.4. The `DOMObject` Type

The `DOMObject` type is used to represent an object.

For Java and ECMAScript, `DOMObject` is bound to the `Object` type.

1.3. General Considerations

1.3.1. String Comparisons in the DOM

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

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

The character normalization, i.e. transforming into their fully normalized form as as defined in [XML 1.1], is assumed to happen at serialization time. The DOM Level 3 Load and Save module [DOM Level 3 Load and Save] provides a serialization mechanism (see the `DOMSerializer` interface, section 2.3.1) and uses the `DOMConfiguration` parameters "" and "" to assure that text is fully normalized [XML 1.1].

Other serialization mechanisms built on top of the DOM Level 3 Core also have to assure that text is *fully normalized*.

1.3.2. DOM URIs

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

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

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

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

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

1.3.3. XML Namespaces

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

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

In general, the DOM implementation (and higher) doesn't perform any URI normalization or canonicalization. The URIs given to the DOM are assumed to be valid (e.g., characters such as white spaces are properly escaped), and no lexical checking is performed. Absolute URI references are treated as strings and . How relative namespace URI references are treated is undefined. To ensure interoperability only absolute namespace URI references (i.e., URI references beginning with a scheme name and a colon) should be used. Applications should use the value `null` as the `namespaceURI` parameter for methods if they wish to have no namespace. In programming languages where empty strings can be differentiated from null, empty strings, when given as a namespace URI, are converted to `null`. This is true even though the DOM does no lexical checking of URIs.

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

☞ 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 or is "xmlns" as introduced in [XML Namespaces 1.1].

In a document with no namespaces, the list of an `EntityReference` node is always the same as that of the corresponding `Entity`. This is not true in a document where an entity contains unbound . In such a case, the of the corresponding `EntityReference` nodes may be bound to different , depending on where the entity references are. Also, because, in the DOM, nodes always remain bound to the same namespace URI, moving such `EntityReference` nodes can lead to documents that cannot be serialized. This is also true when the DOM Level 1 method `Document.createEntityReference(name)` is used to create entity references that correspond to such entities, since the of the returned `EntityReference` are unbound. While DOM Level 3 does have support for the resolution of namespace prefixes, use of such entities and entity references should be avoided or used with extreme care.

The "NS" methods, such as `Document.createElementNS(namespaceURI, qualifiedName)` and `Document.createAttributeNS(namespaceURI, qualifiedName)`, are meant to be used by namespace aware applications. Simple applications that do not use namespaces can use the DOM Level 1 methods, such as `Document.createElement(tagName)` and `Document.createAttribute(name)`. Elements and attributes created in this way do not have any namespace prefix, namespace URI, or local name.

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

1.3.4. Base URIs

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

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

1.3.5. Mixed DOM Implementations

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

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

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

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

1.3.6. DOM Features

Each DOM module defines one or more features, as listed in the conformance section (Preface C.7 – Conformance on page 8). Features are case-insensitive and are also defined for a specific set of versions. For example, this specification defines the features `"Core"` and `"XML"`, for the version `"3.0"`. Versions `"1.0"` and `"2.0"` can also be used for features defined in the corresponding DOM Levels. To avoid possible conflicts, as a convention, names referring to features defined outside the DOM specification should be made unique. Applications could then request for features to be supported by a DOM implementation using the methods `DOMImplementationSource.getDOMImplementation(features)` or `DOMImplementationSource.getDOMImplementationList(features)`, check the features supported by a DOM implementation using the method `DOMImplementation.hasFeature(fea-`

`ture, version)`, or by a specific node using `Node.isSupported(feature, version)`. Note that when using the methods that take a feature and a version as parameters, applications can use `null` or empty string for the version parameter if they don't wish to specify a particular version for the specified feature.

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

As discussed § 1.3.5 – Mixed DOM Implementations on page 16, DOM Level 3 Core should be able to collaborate with subcomponents implementing specific DOMs. For that effect, the methods `DOMImplementation.getFeature(feature, version)` and `Node.getFeature(feature, version)` were introduced. In the case of `DOMImplementation.hasFeature(feature, version)` and `Node.isSupported(feature, version)`, if a plus sign "+" is prepended to any feature name, implementations are considered in which the specified feature may not be directly castable but would require discovery through `DOMImplementation.getFeature(feature, version)` and `Node.getFeature(feature, version)`. Without a plus, only features whose interfaces are directly castable are considered.

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

1.3.7. Bootstrapping

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

To solve these problems this specification introduces a `DOMImplementationRegistry` object with a function that lets an application find implementations, based on the specific features it requires. How this object is found and what it exactly looks like is not defined here, because this cannot be done in a language-independent manner. Instead, each language binding defines its own way of doing this. See Appendix G – Java Language Binding on page 44 and Appendix H – ECMAScript Language Binding on page 47 for specifics.

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

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

1.4. Fundamental Interfaces: Core Module

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

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

1.5. Extended Interfaces: XML Module

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

The interfaces found within this section are not mandatory. A DOM application may use the `DOMImplementation.hasFeature(feature, version)` method with parameter values "XML" and "3.0" (respectively) to determine whether or not this module is supported by the implementation. In order to fully support this module, an implementation must also support the "Core" feature defined in § 1.4 – Fundamental Interfaces: Core Module on page 18 and the feature "XMLVersion" with version "1.0" defined in `Document.xmlVersion`. Please refer to additional information about Preface C.7 – Conformance on page 8 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`.

Appendix A. Changes

Philippe Le Hégaret, W3C
This section summarizes the changes between [DOM Level 2 Core] and this new version of the Core specification.

A.1. New sections

The following new sections have been added:

- Preface C.6 – DOM Architecture on page 7: a global overview of the DOM Level 3 modules;
- § 1.3.2 – DOM URIs on page 14: general considerations on the URI handling in DOM Level 3;
- § 1.3.4 – Base URIs on page 15: How the [base URI] property defined in [XML Information Set] has been exposed in DOM Level 3;
- § 1.3.5 – Mixed DOM Implementations on page 16: general considerations on DOM implementation extensions;
- § 1.3.6 – DOM Features on page 16: overview of the DOM features and how they relate to the DOM modules;
- § 1.3.7 – Bootstrapping on page 17: general introduction to the DOM Level 3 bootstrapping mechanisms;
- Appendix B – Namespaces Algorithms on page 21: how namespace URIs and prefixes are resolved in DOM Level 3;
- Appendix C – Infoset Mapping on page 31: relation between DOM Level 3 and [XML Information Set];
- Appendix D – Configuration Settings on page 42: relations between parameters as used in `DOMConfiguration`;

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

***Interface* `Attr`**

The `Attr` interface has two new attributes, `Attr.schemaTypeInfo`, and `Attr.isId`.

***Interface* `Document`**

The `Document` interface has seven new attributes: `Document.inputEncoding`, `Document.xmlEncoding`, `Document.xmlStandalone`, `Document.xmlVersion`, `Document.strictErrorChecking`, `Document.documentURI`, and `Document.domConfig`. It has three new methods: `Document.adoptNode(source)`, `Document.normalizeDocument()`, and `Document.renameNode(n, namespaceURI, qualifiedName)`. The attribute `Document.doctype` has been modified.

***Exception* `DOMException`**

The `DOMException` has two new exception codes: `VALIDATION_ERR` and `TYPE_MISMATCH_ERR`.

***Interface* `DOMImplementation`**

The `DOMImplementation` interface has one new method, `DOMImplementation.getFeature(feature, version)`.

***Interface* `Entity`**

The `Entity` interface has three new attributes: `Entity.inputEncoding`, `Entity.xmlEncoding`, and `Entity.xmlVersion`.

***Interface* `Element`**

The `Element` interface has one new attribute, `Element.schemaTypeInfo`, and three new methods: `Element.setIdAttribute(name, isId)`, `Element.setIdAttributeNS(namespaceURI, localName, isId)`, and `Element.setIdAttributeNode(idAttr, isId)`.

Interface `Node`

The `Node` interface has two new attributes, `Node.baseURI` and `Node.textContent`. It has nine new methods: `Node.compareDocumentPosition(other)`, `Node.isSameNode(other)`, `Node.lookupPrefix(namespaceURI)`, `Node.isDefaultNamespace(namespaceURI)`, `Node.lookupNamespaceURI(prefix)`, `Node.isEqualNode(arg)`, `Node.getFeature(feature, version)`, `Node.setUserData(key, data, handler)`, `Node.getUserData(key)`. It introduced 6 new constants: `Node.DOCUMENT_POSITION_DISCONNECTED`, `Node.DOCUMENT_POSITION_PRECEDING`, `Node.DOCUMENT_POSITION_FOLLOWING`, `Node.DOCUMENT_POSITION_CONTAINS`, `Node.DOCUMENT_POSITION_CONTAINED_BY`, and `Node.DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC`. The methods `Node.insertBefore(newChild, refChild)`, `Node.replaceChild(newChild, oldChild)` and `Node.removeChild(oldChild)` have been modified.

Interface `Text`

The `Text` interface has two new attributes, `Text.wholeText` and `Text.isElementContentWhitespace`, and one new method, `Text.replaceWholeText(content)`.

A.3. New DOM features

"XMLVersion"

The "XMLVersion" DOM feature was introduced to represent if an implementation is able to support [XML 1.0] or [XML 1.1]. See `Document.xmlVersion`.

A.4. New types

`DOMUserData`

The `DOMUserData` type was added to the Core module.

`DOMObject`

The `DOMObject` type was added to the Core module.

A.5. New interfaces

`DOMStringList`

The `DOMStringList` interface has one attribute, `DOMStringList.length`, and one method, `DOMStringList.item(index)`.

`NameList`

The `NameList` interface has one attribute, `NameList.length`, and two methods, `NameList.getName(index)` and `NameList.getNamespaceURI(index)`.

`DOMImplementationList`

The `DOMImplementationList` interface has one attribute, `DOMImplementationList.length`, and one method, `DOMImplementationList.item(index)`.

DOMImplementationSource

The `DOMImplementationSource` interface has two methods, `DOMImplementationSource.getDOMImplementation(features)`, and `DOMImplementationSource.getDOMImplementationList(features)`.

TypeInfo

The `TypeInfo` interface has two attributes, `TypeInfo.typeName`, and `TypeInfo.typeNamespace`.

UserDataHandler

The `UserDataHandler` interface has one method, `UserDataHandler.handle(operation, key, data, src, dst)`, and four constants: `UserDataHandler.NODE_CLONED`, `UserDataHandler.NODE_IMPORTED`, `UserDataHandler.NODE_DELETED`, and `UserDataHandler.NODE_RENAMED`.

DOMError

The `DOMError` interface has six attributes: `DOMError.severity`, `DOMError.message`, `DOMError.type`, `DOMError.relatedException`, `DOMError.relatedData`, and `DOMError.location`. It has four constants: `DOMError.SEVERITY_WARNING`, `DOMError.SEVERITY_ERROR`, and `DOMError.SEVERITY_FATAL_ERROR`.

DOMErrorHandler

The `DOMErrorHandler` interface has one method: `DOMErrorHandler.handleError(error)`.

DOMLocator

The `DOMLocator` interface has seven attributes: `DOMLocator.lineNumber`, `DOMLocator.columnNumber`, `DOMLocator.byteOffset`, `DOMLocator.utf16Offset`, `DOMLocator.relatedNode`, `DOMLocator.uri`, and `DOMLocator.lineNumber`.

DOMConfiguration

The `DOMConfiguration` interface has one attribute: `DOMConfiguration.parameterNames`. It also has three methods: `DOMConfiguration.setParameter(name, value)`, `DOMConfiguration.getParameter(name)`, and `DOMConfiguration.canSetParameter(name, value)`.

A.6. Objects

This specification defines one object, only provided in the bindings:

DOMImplementationRegistry

The `DOMImplementationRegistry` object has two methods, `DOMImplementationRegistry.getDOMImplementation(features)`, and `DOMImplementationRegistry.getDOMImplementationList(features)`.

Appendix B. Namespaces Algorithms

Arnaud Le Hors, IBM; Elena Litani, IBM

This appendix contains several namespace algorithms, such as namespace normalization algorithm that fixes namespace information in the Document Object Model to produce a document. If [XML 1.0] is in use (see `Document.xmlVersion`) the algorithms conform to [XML Namespaces], otherwise if [XML 1.1] is in use, algorithms conform to [XML Namespaces 1.1].

B.1. Namespace Normalization

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

```
void Element.normalizeNamespaces()
{

  // Pick up local namespace declarations
  //
  for ( all DOM Level 2 valid local namespace declaration attributes of Element )
  {
      if (the namespace declaration is invalid)
      {
          // Note: The prefix xmlns is used only to declare namespace bindings and
          // is by definition bound to the namespace name
http://www.w3.org/2000/xmlns/.
         // It must not be declared. No other prefix may be bound to this namespace
 name.

          ==> Report an error.

      }
      else
      {
          ==>  Record the namespace declaration
      }
  }


  // Fixup element's namespace
  //
  if ( Element's namespaceURI != null )
  {
    if ( Element's prefix/namespace pair (or default namespace,
         if no prefix) are within the scope of a binding )
    {
      ==> do nothing, declaration in scope is inherited

      See section "B.1.1: Scope of a binding" for an example

    }
    else
    {
      ==> Create a local namespace declaration attr for this namespace,
          with Element's current prefix (or a default namespace, if
          no prefix). If there's a conflicting local declaration
          already present, change its value to use this namespace.

          See section "B.1.2: Conflicting namespace declaration" for an example

          // NOTE that this may break other nodes within this Element's
          // subtree, if they're already using this prefix.
          // They will be repaired when we reach them.
    }
  }
  else
  {
    // Element has no namespace URI:
    if ( Element's localName is null )
    {
       // DOM Level 1 node
       ==> if in process of validation against a namespace aware schema
           (i.e XML Schema) report a fatal error: the processor can not recover
           in this situation.
           Otherwise, report an error: no namespace fixup will be performed on this
node.
    }
    else
    {
      // Element has no pseudo-prefix
      if ( there's a conflicting local default namespace declaration
           already present )
      {
        ==> change its value to use this empty namespace.
      }
      // NOTE that this may break other nodes within this Element's
      // subtree, if they're already using the default namespaces.
      // They will be repaired when we reach them.
    }
  }


  // Examine and polish the attributes
  //
  for ( all non-namespace Attrs of Element )
  {
     if ( Attr[i] has a namespace URI )
     {
        if ( attribute has no prefix (default namespace decl does not apply to
attributes)
             OR
             attribute prefix is not declared
             OR
             conflict: attribute has a prefix that conflicts with a binding
                       already active in scope)
        {
          if (namespaceURI matches an in scope declaration of one or more prefixes)

           {
               // pick the most local binding available;
               // if there is more than one pick one arbitrarily

               ==> change attribute's prefix.
           }
           else
           {
              if (the current prefix is not null and it has no in scope declaration)

               {
                   ==> declare this prefix
               }
               else
               {
                   // find a prefix following the pattern "NS" +index (starting at
1)
                   // make sure this prefix is not declared in the current scope.
                   // create a local namespace declaration attribute

                   ==> change attribute's prefix.
               }
           }
        }
     }
     else
     {
        // Attr[i] has no namespace URI

        if ( Attr[i] has no localName )
        {
           // DOM Level 1 node
           ==> if in process of validation against a namespace aware schema
```

```
               (i.e XML Schema) report a fatal error: the processor can not recover

                in this situation.
                Otherwise, report an error: no namespace fixup will be performed on
 this node.
        }
        else
        {
           // attr has no namespace URI and no prefix
           // no action is required, since attrs don't use default
           ==> do nothing
        }
     }
  } // end for-all-Attrs

  // do this recursively
  for ( all child elements of Element )
  {
    childElement.normalizeNamespaces()
  }
} // end Element.normalizeNamespaces
```

B.1.1. Scope of a Binding

☞ This section is informative.

An element's prefix/namespace URI pair is said to be within the scope of a binding if its namespace prefix is bound to the same namespace URI in the [in-scope namespaces] defined in [XML Information Set].

As an example, the following document is loaded in a DOM tree:

```
<root>
  <parent xmlns:ns="http://www.example.org/ns1"
          xmlns:bar="http://www.example.org/ns2">
    <ns:child1 xmlns:ns="http://www.example.org/ns2"/>
  </parent>
</root>
```

In the case of the `child1` element, the namespace prefix and namespace URI are within the scope of the appropriate namespace declaration given that the namespace prefix `ns` of `child1` is bound to `http://www.example.org/ns2`.

Using the method `Node.appendChild`, a `child2` element is added as a sibling of `child1` with the same namespace prefix and namespace URI, i.e. `"ns"` and `"http://www.example.org/ns2"` respectively. Unlike `child1` which contains the appropriate namespace declaration in its attributes, `child2`'s prefix/namespace URI pair is within the scope of the namespace declaration of its parent, and the namespace prefix `"ns"` is bound to `"http://www.example.org/ns1"`. `child2`'s prefix/namespace URI pair is therefore not within the scope of a binding. In order to put them within a scope of a binding, the namespace normalization algorithm will create a namespace declaration attribute value to bind the namespace prefix `"ns"` to the namespace URI `"http://www.example.org/ns2"` and will attach to `child2`. The XML representation of the document after the completion of the namespace normalization algorithm will be:

```
<root>
  <parent xmlns:ns="http://www.example.org/ns1"
          xmlns:bar="http://www.example.org/ns2">
    <ns:child1 xmlns:ns="http://www.example.org/ns2"/>
    <ns:child2 xmlns:ns="http://www.example.org/ns2"/>
  </parent>
</root>
```

To determine if an element is within the scope of a binding, one can invoke `Node.lookupNamespaceURI`, using its namespace prefix as the parameter, and compare the resulting namespace URI against the desired URI, or one can invoke `Node.isDefaultNamespaceURI` using its namespace URI if the element has no namespace prefix.

B.1.2. Conflicting Namespace Declaration

☞ This section is informative.

A conflicting namespace declaration could occur on an element if an `Element` node and a namespace declaration attribute use the same prefix but map them to two different namespace URIs.

As an example, the following document is loaded in a DOM tree:

```
<root>
  <ns:child1 xmlns:ns="http://www.example.org/ns1">
    <ns:child2/>
  </ns:child1>
</root>
```

Using the method `Node.renameNode`, the namespace URI of the element `child1` is renamed from `"http://www.example.org/ns1"` to `"http://www.example.org/ns2"`. The namespace prefix `"ns"` is now mapped to two different namespace URIs at the element `child1` level and thus the namespace declaration is declared conflicting. The namespace normalization algorithm will resolved the namespace prefix conflict by modifying the namespace declaration attribute value from `"http://www.example.org/ns1"` to `"http://www.example.org/ns2"`. The algorithm will then continue and consider the element `child2`, will no longer find a namespace declaration mapping the namespace prefix `"ns"` to `"http://www.example.org/ns1"` in the element's scope, and will create a new one. The XML representation of the document after the completion of the namespace normalization algorithm will be:

```
<root>
  <ns:child1 xmlns:ns="http://www.example.org/ns2">
```

```
    <ns:child2  xmlns:ns="http://www.example.org/ns1"/>
  </ns:child1>
</root>
```

B.2. Namespace Prefix Lookup

The following describes in pseudo code the algorithm used in the `lookupPrefix` method of the `Node` interface. Before returning found prefix the algorithm needs to make sure that the prefix is not redefined on an element from which the lookup started. This methods ignores DOM Level 1 nodes.

☞ This method ignores all default namespace declarations. To look up default namespace use `isDefaultNamespace` method.

```
DOMString lookupPrefix(in DOMString namespaceURI)
{
  if (namespaceURI has no value, i.e. namespaceURI is null or empty string) {
     return null;
  }
  short type = this.getNodeType();
  switch (type) {
        case Node.ELEMENT_NODE:
        {
             return lookupNamespacePrefix(namespaceURI, this);
        }
        case Node.DOCUMENT_NODE:
        {
             return getDocumentElement().lookupNamespacePrefix(namespaceURI);
        }
        case Node.ENTITY_NODE :
        case Node.NOTATION_NODE:
        case Node.DOCUMENT_FRAGMENT_NODE:
        case Node.DOCUMENT_TYPE_NODE:
            return null;  // type is unknown
        case Node.ATTRIBUTE_NODE:
        {
             if ( Attr has an owner Element )
             {
                 return ownerElement.lookupNamespacePrefix(namespaceURI);
             }
             return null;
        }
        default:
        {
           if (Node has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
           {
                    return ancestor.lookupNamespacePrefix(namespaceURI);
           }
            return null;
        }
     }
 }


DOMString lookupNamespacePrefix(DOMString namespaceURI, Element originalElement){
        if ( Element has a namespace and Element's namespace == namespaceURI and
             Element has a prefix and
             originalElement.lookupNamespaceURI(Element's prefix) == namespaceURI)

        {
             return (Element's prefix);
        }
        if ( Element has attributes)
        {
            for ( all DOM Level 2 valid local namespace declaration attributes of
Element )
            {
                if (Attr's prefix == "xmlns" and
                   Attr's value == namespaceURI and
                   originalElement.lookupNamespaceURI(Attr's localname) ==
namespaceURI)
                   {
                      return (Attr's localname);
                   }
            }
        }

        if (Node has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
        {
            return ancestor.lookupNamespacePrefix(namespaceURI, originalElement);
        }
        return null;
    }
```

B.3. Default Namespace Lookup

The following describes in pseudo code the algorithm used in the `isDefaultNamespace` method of the `Node` interface. This methods ignores DOM Level 1 nodes.

```
boolean isDefaultNamespace(in DOMString namespaceURI)
{
  switch (nodeType) {
  case ELEMENT_NODE:
     if ( Element has no prefix )
     {
          return (Element's namespace == namespaceURI);
     }
     if ( Element has attributes and there is a valid DOM Level 2
          default namespace declaration, i.e. Attr's localName == "xmlns" )
     {
   return (Attr's value == namespaceURI);
     }

     if ( Element has an ancestor Element )
         // EntityReferences may have to be skipped to get to it
     {
          return ancestorElement.isDefaultNamespace(namespaceURI);
     }
     else {
          return unknown (false);
     }
  case DOCUMENT_NODE:
     return documentElement.isDefaultNamespace(namespaceURI);
  case ENTITY_NODE:
  case NOTATION_NODE:
  case DOCUMENT_TYPE_NODE:
  case DOCUMENT_FRAGMENT_NODE:
     return unknown (false);
  case ATTRIBUTE_NODE:
     if ( Attr has an owner Element )
     {
          return ownerElement.isDefaultNamespace(namespaceURI);
     }
     else {
          return unknown (false);
     }
  default:
     if ( Node has an ancestor Element )
         // EntityReferences may have to be skipped to get to it
     {
          return ancestorElement.isDefaultNamespace(namespaceURI);
     }
     else {
          return unknown (false);
     }
  }
}
```

B.4. Namespace URI Lookup

The following describes in pseudo code the algorithm used in the `lookupNamespaceURI` method of the `Node` interface. This methods ignores DOM Level 1 nodes.

```
DOMString lookupNamespaceURI(in DOMString prefix)
{
  switch (nodeType) {
     case ELEMENT_NODE:
     {
         if ( Element's namespace != null and Element's prefix == prefix )
         {
              // Note: prefix could be "null" in this case we are looking for default
 namespace
                return (Element's namespace);
         }
         if ( Element has attributes)
         {
            for ( all DOM Level 2 valid local namespace declaration attributes of
Element )
            {
                 if (Attr's prefix == "xmlns" and Attr's localName == prefix )
                       // non default namespace
                 {
                        if (Attr's value is not empty)
                        {
                          return (Attr's value);
                        }
                        return unknown (null);
                 }
                 else if (Attr's localname == "xmlns" and prefix == null)
                       // default namespace
                 {
                        if (Attr's value is not empty)
                        {
                          return (Attr's value);
                        }
                        return unknown (null);
                 }
           }
         }
```

```
        if ( Element has an ancestor Element )
           // EntityReferences may have to be skipped to get to it
        {
                  return ancestorElement.lookupNamespaceURI(prefix);
        }
        return null;
    }
    case DOCUMENT_NODE:
         return documentElement.lookupNamespaceURI(prefix)

    case ENTITY_NODE:
    case NOTATION_NODE:
    case DOCUMENT_TYPE_NODE:
    case DOCUMENT_FRAGMENT_NODE:
          return unknown (null);

    case ATTRIBUTE_NODE:
        if (Attr has an owner Element)
        {
            return ownerElement.lookupNamespaceURI(prefix);
        }
        else
        {
            return unknown (null);
        }
    default:
        if (Node has an ancestor Element)
         // EntityReferences may have to be skipped to get to it
        {
            return ancestorElement.lookupNamespaceURI(prefix);
        }
        else {
            return unknown (null);
        }
  }
}
```

Appendix C. Infoset Mapping

Philippe Le Hégaret, W3C

This appendix contains the mappings between the XML Information Set [XML Information Set] model and the Document Object Model. Starting from a `Document` node, each *information item* is mapped to its respective `Node`, and each `Node` is mapped to its respective *information item*. As used in the Infoset specification, the Infoset property names are shown in square brackets, [thus].

Unless specified, the Infoset to DOM node mapping makes no distinction between unknown and no value since both will be exposed as `null` (or `false` if the DOM attribute is of type `boolean`).

C.1. Document Node Mapping

C.1.1. Infoset to Document Node

An *document information item* maps to a `Document` node. The attributes of the corresponding `Document` node are constructed as follows:

Attribute	Value
`Node.nodeName`	`"#document"`
`Node.nodeValue`	`null`
`Node.nodeType`	`Node.DOCUMENT_NODE`
`Node.parentNode`	`null`
`Node.childNodes`	A `NodeList` containing the information items in the [children] property.
`Node.firstChild`	The first node contained in `Node.childNodes`
`Node.lastChild`	The last node contained in `Node.childNodes`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	`null`
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	same as `Document.documentURI`
`Node.textContent`	`null`
`Document.doctype`	The document type information item
`Document.implementation`	The `DOMImplementation` object used to create this node
`Document.documentElement`	The [document element] property
`Document.inputEncoding`	The [character encoding scheme] property
`Document.xmlEncoding`	`null`
`Document.xmlStandalone`	The [standalone] property, or `false` if the latter has no value.
`Document.xmlVersion`	The [version] property, or `"1.0"` if the latter has no value.
`Document.strictErrorChecking`	`true`
`Document.documentURI`	The [base URI] property
`Document.domConfig`	A `DOMConfiguration` object whose parameters are set to their default values

The [notations], [unparsed entities] properties are being exposed in the `DocumentType` node.

☞ The [all declarations processed] property is not exposed through the `Document` node.

C.1.2. Document Node to Infoset

A `Document` node maps to an *document information item*. `Document` nodes with no namespace URI (`Node.namespaceURI` equals to `null`) cannot be represented using the Infoset. The properties of the corresponding *document information item* are constructed as follows:

Property	Value
[children]	`Node.childNodes`
[document element]	`Document.documentElement`
[notations]	`Document.doctype.notations`
[unparsed entities]	The information items from `Document.doctype.entities`, whose `Node.childNodes` is an empty list
[base URI]	`Document.documentURI`
[character encoding scheme]	`Document.inputEncoding`
[standalone]	`Document.xmlStandalone`
[version]	`Document.xmlVersion`
[all declarations processed]	The value is implementation dependent

C.2. Element Node Mapping

C.2.1. Infoset to Element Node

An *element information item* maps to a `Element` node. The attributes of the corresponding `Element` node are constructed as follows:

Attribute	Value
`Node.nodeName`	same as `Element.tagName`
`Node.nodeValue`	`null`
`Node.nodeType`	`Node.ELEMENT_NODE`
`Node.parentNode`	The [parent] property
`Node.childNodes`	A `NodeList` containing the information items in the [children] property
`Node.firstChild`	The first node contained in `Node.childNodes`
`Node.lastChild`	The last node contained in `Node.childNodes`
`Node.previousSibling`	The information item preceding the current one on the [children] property contained in the [parent] property
`Node.nextSibling`	The information item following the current one on the [children] property contained in the [parent] property
`Node.attributes`	The information items contained in the [attributes] and [namespace attributes] properties
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	The [namespace name] property
`Node.prefix`	The [prefix] property
`Node.localName`	The [local name] property
`Node.baseURI`	The [base URI] property
`Node.textContent`	Concatenation of the `Node.textContent` attribute value of every child node, excluding `COMMENT_NODE` and `PROCESSING_INSTRUCTION_NODE` nodes. This is the empty string if the node has no children.

`Element.tagName`	If the [prefix] property has no value, this contains the [local name] property. Otherwise, this contains the concatenation of the [prefix] property, the colon ':' character, and the [local name] property.
`Element.schemaTypeInfo`	A `TypeInfo` object whose `TypeInfo.typeNamespace` and `TypeInfo.typeName` are inferred from the schema in use if available.

☞ The [in-scope namespaces] property is not exposed through the `Element` node.

C.2.2. Element Node to Infoset

An `Element` node maps to an *element information item*. Because the Infoset only represents unexpanded entity references, non-empty `EntityReference` nodes contained in `Node.childNodes` need to be replaced by their content. DOM applications could use the `Document.normalizeDocument()` method for that effect with the "" parameter set to `false`. The properties of the corresponding *element information item* are constructed as follows:

Property	Value
[namespace name]	`Node.namespaceURI`
[local name]	`Node.localName`
[prefix]	`Node.prefix`
[children]	`Node.childNodes`, whose expanded entity references (`EntityReference` nodes with children) have been replaced with their content.
[attributes]	The nodes contained in `Node.attributes`, whose `Node.namespaceURI` value is different from `"http://www.w3.org/2000/xmlns/"`
[namespace attributes]	The nodes contained in `Node.attributes`, whose `Node.namespaceURI` value is `"http://www.w3.org/2000/xmlns/"`
[in-scope namespaces]	The namespace information items computed using the [namespace attributes] properties of this node and its ancestors. If the [DOM Level 3 XPath] module is supported, the namespace information items can also be computed from the `XPathNamespace` nodes.
[base URI]	`Node.baseURI`
[parent]	`Node.parentNode`

C.3. Attr Node Mapping

C.3.1. Infoset to Attr Node

An *attribute information item* map to a `Attr` node. The attributes of the corresponding `Attr` node are constructed as follows:

Attribute/Method	Value
`Node.nodeName`	same as `Attr.name`
`Node.nodeValue`	same as `Attr.value`
`Node.nodeType`	`Node.ATTRIBUTE_NODE`
`Node.parentNode`	`null`
`Node.childNodes`	A `NodeList` containing one `Text` node whose text content is the same as `Attr.value`.
`Node.firstChild`	The `Text` node contained in `Node.childNodes`
`Node.lastChild`	The `Text` node contained in `Node.childNodes`

`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	The [namespace name] property
`Node.prefix`	The [prefix] property
`Node.localName`	The [local name] property
`Node.baseURI`	`null`
`Node.textContent`	the value of `Node.textContent` of the `Text` child. same as `Node.nodeValue` (since this attribute node only contains one `Text` node)
`Attr.name`	If the [prefix] property has no value, this contains the [local name] property. Otherwise, this contains the concatenation of the [prefix] property, the colon ':' character, and the [local name] property.
`Attr.specified`	The [specified] property
`Attr.value`	The [normalized value] property
`Attr.ownerElement`	The [owner element] property
`Attr.schemaTypeInfo`	A `TypeInfo` object whose `TypeInfo.typeNamespace` is `"http://www.w3.org/TR/REC-xml"` and `TypeInfo.typeName` is the [attribute type] property
`Attr.isId`	if the [attribute type] property is ID, this method return `true`

C.3.2. Attr Node to Infoset

An `Attr` node maps to an *attribute information item*. `Attr` nodes with no namespace URI (`Node.namespaceURI` equals to `null`) cannot be represented using the Infoset. The properties of the corresponding *attribute information item* are constructed as follows:

Property	Value
[namespace name]	`Node.namespaceURI`
[local name]	`Node.localName`
[prefix]	`Node.prefix`
[normalized value]	`Attr.value`
[specified]	`Attr.specified`
[attribute type]	Using the `TypeInfo` object referenced from `Attr.schemaTypeInfo`, the value of `TypeInfo.typeName` if `TypeInfo.typeNamespace` is `"http://www.w3.org/TR/REC-xml"`.
[references]	if the computed [attribute type] property is IDREF, IDREFS, ENTITY, ENTITIES, or NOTATION, the value of this property is an ordered list of the element, unparsed entity, or notation information items referred to in the attribute value, in the order that they appear there. The ordered list is computed using `Node.ownerDocument.getElementById`, `Node.ownerDocument.doctype.entities`, and `Node.ownerDocument.doctype.notations`.
[owner element]	`Attr.ownerElement`

C.4. ProcessingInstruction Node Mapping

C.4.1. Infoset to ProcessingInstruction Node

A *processing instruction information item* map to a `ProcessingInstruction` node. The attributes of the corresponding `ProcessingInstruction` node are constructed as follows:

Attribute	Value
`Node.nodeName`	same as `ProcessingInstruction.target`
`Node.nodeValue`	same as `ProcessingInstruction.data`
`Node.nodeType`	`Node.PROCESSING_INSTRUCTION_NODE`
`Node.parentNode`	The [parent] property
`Node.childNodes`	empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	The [base URI] property of the parent element if any. The [base URI] property of the processing instruction information item is not exposed through the `ProcessingInstruction` node.
`Node.textContent`	same as `Node.nodeValue`
`ProcessingInstruction.target`	The [target] property
`ProcessingInstruction.data`	The [content] property

C.4.2. ProcessingInstruction Node to Infoset

A `ProcessingInstruction` node maps to an *processing instruction information item*. The properties of the corresponding *processing instruction information item* are constructed as follows:

Property	Value
[target]	`ProcessingInstruction.target`
[content]	`ProcessingInstruction.data`
[base URI]	`Node.baseURI` (which is equivalent to the base URI of its parent element if any)
[notation]	The `Notation` node named by the target and if available from `Node.ownerDocument.doctype.notations`
[parent]	`Node.parentNode`

C.5. EntityReference Node Mapping

C.5.1. Infoset to EntityReference Node

An *unexpanded entity reference information item* maps to a `EntityReference` node. The attributes of the corresponding `EntityReference` node are constructed as follows:

Attribute	Value
`Node.nodeName`	The [name] property
`Node.nodeValue`	`null`
`Node.nodeType`	`Node.ENTITY_REFERENCE_NODE`
`Node.parentNode`	the [parent] property
`Node.childNodes`	Empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	The [declaration base URI] property
`Node.textContent`	`null` (the node has no children)

☞ The [system identifier] and [public identifier] properties are not exposed through the `EntityReference` node, but through the `Entity` node reference from this `EntityReference` node, if any.

C.5.2. EntityReference Node to Infoset

An `EntityReference` node maps to an *unexpanded entity reference information item*. `EntityReference` nodes with children (`Node.childNodes` contains a non-empty list) cannot be represented using the Infoset. The properties of the corresponding *unexpanded entity reference information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName`
[system identifier]	The `Entity.systemId` value of the `Entity` node available from `Node.ownerDocument.doctype.entities` if available
[public identifier]	The `Entity.publicId` value of the `Entity` node available from `Node.ownerDocument.doctype.entities` if available
[declaration base URI]	`Node.baseURI`
[parent]	`Node.parentNode`

C.6. Text and CDATASection Nodes Mapping

Since the [XML Information Set] doesn't represent the boundaries of CDATA marked sections, `CDATASection` nodes cannot occur from an infoset mapping.

C.6.1. Infoset to Text Node

Consecutive *character information items* map to a `Text` node. The attributes of the corresponding `Text` node are constructed as follows:

Attribute/Method	Value
`Node.nodeName`	`"#text"`
`Node.nodeValue`	same as `CharacterData.data`
`Node.nodeType`	`Node.TEXT_NODE`
`Node.parentNode`	The [parent] property
`Node.childNodes`	empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	null
`Node.textContent`	same as `Node.nodeValue`
`CharacterData.data`	A `DOMString` including all [character code] contained in the *character information items*
`CharacterData.length`	The number of 16-bit units needed to encode all ISO 10646 character code contained in the *character information items* using the UTF-16 encoding.
`Text.isElementContentWhitespace`	The [element content whitespace] property
`Text.wholeText`	same as `CharacterData.data`

☞ By construction, the values of the [parent] and [element content whitespace] properties are necessarily the sames for all consecutive *character information items*.

C.6.2. Text and CDATASection Nodes to Infoset

The text content of a `Text` or a `CDATASection` node maps to a sequence of *character information items*. The number of items is less or equal to `CharacterData.length`. Text nodes contained in `Attr` nodes are mapped to the Infoset using the `Attr.value` attribute. Text nodes contained in `Document` nodes cannot be represented using the Infoset. The properties of the corresponding *character information items* are constructed as follows:

Property	Value
[character code]	The ISO 10646 character code produced using one or two *16-bit units* from `CharacterData.data`
[element content whitespace]	`Text.isElementContentWhitespace`
[parent]	`Node.parentNode`

C.7. Comment Node Mapping

C.7.1. Infoset to Comment Node

A *comment information item* maps to a `Comment` node. The attributes of the corresponding `Comment` node are constructed as follows:

Attribute	Value
`Node.nodeName`	`"#comment"`
`Node.nodeValue`	same as `CharacterData.data`
`Node.nodeType`	`Node.COMMENT_NODE`
`Node.parentNode`	The [parent] property
`Node.childNodes`	empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	`null`
`Node.textContent`	same as `Node.nodeValue`
`CharacterData.data`	The [content] property encoded using the UTF-16 encoding.
`CharacterData.length`	The number of 16-bit units needed to encode all ISO character code contained in the [content] property using the UTF-16 encoding.

C.7.2. Comment Node to Infoset

A `Comment` maps to a *comment information item*. The properties of the corresponding *comment information item* are constructed as follows:

Property	Value
[content]	`CharacterData.data`
[parent]	`Node.parentNode`

C.8. DocumentType Node Mapping

C.8.1. Infoset to DocumentType Node

A *document type declaration information item* maps to a `DocumentType` node. The attributes of the corresponding `DocumentType` node are constructed as follows:

Attribute	Value
`Node.nodeName`	same as `DocumentType.name`

`Node.nodeValue`	`null`
`Node.nodeType`	`Node.DOCUMENT_TYPE_NODE`
`Node.parentNode`	The [parent] property
`Node.childNodes`	empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	`null`
`Node.textContent`	`null`
`DocumentType.name`	The name of the document element.
`DocumentType.entities`	The [unparsed entities] property available from the document information item.
`DocumentType.notations`	The [notations] property available from the document information item.
`DocumentType.publicId`	The [public identifier] property
`DocumentType.systemId`	The [system identifier] property
`DocumentType.internalSubset`	The value is implementation dependent

The [children] property is not exposed through the `DocumentType` node.

C.8.2. DocumentType Node to Infoset

A `DocumentType` maps to a *document type declaration information item*. The properties of the corresponding *document type declaration information item* are constructed as follows:

Property	Value
[system identifier]	`DocumentType.systemId`
[public identifier]	`DocumentType.publicId`
[children]	The value of this property is implementation dependent
[parent]	`Node.parentNode`

C.9. Entity Node Mapping

C.9.1. Infoset to Entity Node

An *unparsed entity information item* maps to a `Entity` node. The attributes of the corresponding `Entity` node are constructed as follows:

Attribute	Value
`Node.nodeName`	The [name] property
`Node.nodeValue`	`null`

`Node.nodeType`	`Node.ENTITY_NODE`
`Node.parentNode`	`null`
`Node.childNodes`	Empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	The [declaration base URI] property
`Node.textContent`	`""` (the node has no children)
`Entity.publicId`	The [public identifier] property
`Entity.systemId`	The [system identifier] property
`Entity.notationName`	The [notation name] property
`Entity.inputEncoding`	`null`
`Entity.xmlEncoding`	`null`
`Entity.xmlVersion`	`null`

The [notation] property is available through the `DocumentType` node.

C.9.2. Entity Node to Infoset

An `Entity` node maps to an *unparsed entity information item*. `Entity` nodes with children (`Node.childNodes` contains a non-empty list) cannot be represented using the Infoset. The properties of the corresponding *unparsed entity information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName`
[system identifier]	`Entity.systemId`
[public identifier]	`Entity.publicId`
[declaration base URI]	`Node.baseURI`
[notation name]	`Entity.notationName`
[notation]	The `Notation` node referenced from `DocumentType.notations` whose name is the [notation name] property

C.10. Notation Node Mapping

C.10.1. Infoset to Notation Node

A *notation information item* maps to a `Notation` node. The attributes of the corresponding `Notation` node are constructed as follows:

Attribute	Value
`Node.nodeName`	The [name] property
`Node.nodeValue`	`null`
`Node.nodeType`	`Node.NOTATION_NODE`
`Node.parentNode`	`null`
`Node.childNodes`	Empty `NodeList`
`Node.firstChild`	`null`
`Node.lastChild`	`null`
`Node.previousSibling`	`null`
`Node.nextSibling`	`null`
`Node.attributes`	`null`
`Node.ownerDocument`	The document information item
`Node.namespaceURI`	`null`
`Node.prefix`	`null`
`Node.localName`	`null`
`Node.baseURI`	The [declaration base URI] property
`Node.textContent`	`null`
`Notation.publicId`	The [public identifier] property
`Notation.systemId`	The [system identifier] property

C.10.2. Notation Node to Infoset

A `Notation` maps to a *notation information item*. The properties of the corresponding *notation information item* are constructed as follows:

Property	Value
[name]	`Node.nodeName`
[system identifier]	`Notation.systemId`
[public identifier]	`Notation.publicId`
[parent]	`Node.parentNode`

Appendix D. Configuration Settings

Elena Litani, IBM

D.1. Configuration Scenarios

Using the `DOMConfiguration` users can change behavior of the `DOMParser`, `DOMSerializer` and `Document.normalizeDocument()`. If a DOM implementation supports XML Schemas and DTD validation, the table below defines behavior of such implementation following various parameter settings on the `DOMConfiguration`. Errors are effectively reported only if a `DOMErrorHandler` object is attached to the "" parameter.

`""`	`""`	`""`	**Instance schemas, i.e. the current schema**	**Outcome**	**Other parameters**
`null`	`true`	`false`	DTD and XML Schema	Implementation dependent	The outcome of setting the "", "" or "" parameters to `true` or `false` is implementation dependent.
	`false`	`true`			
`null`	`true`	`false`	none	Report an error	Setting the "" to `true` or `false` has no effect on the DOM.
	`false`	`true`		No error is reported	
`null`	`true`	`false`	DTD	Validate against DTD	Setting the "" to `true` or `false` has no effect on the DOM.
	`false`	`true`			
`null`	`true`	`false`	XML Schema	Validate against XML Schema	The outcome of setting the "" to `false` is implementation dependent (likely to be an error). Setting the "" to `false` does not have any effect on the DOM.
	`false`	`true`			
`"http://www.w3.org/TR/REC-xml"`	`true`	`false`	DTD or XML Schema or both	If DTD is found, validate against DTD. Otherwise, report an error.	Setting the "" to `true` or `false` has no effect on the DOM.
	`false`	`true`		If DTD is found, validate against DTD.	
`"http://www.w3.org/2001/XMLSchema"`	`true`	`false`	DTD or XML Schema or both	If XML Schema is found, validate against the schema. Otherwise, report an error.	Setting the "" to `true` exposes XML Schema normalized values in the DOM. The outcome of setting the "" to `false` is implementation dependent (likely to be an error).
	`false`	`true`		If XML Schema is found, validate against the schema.	
`"http://www.w3.org/2001/XMLSchema"` or `"http://www.w3.org/TR/REC-xml"`	`false`	`false`	DTD or XML Schema or both	If XML Schema is found, it is ignored. DOM implementations may use information available in the DTD to perform entity resolution.	Setting the "" to `true` of `false` has no effect on the DOM.

If an error has to be reported, as specified in the "Outcome" column above, the `DOMError.type` is `"no-schema-available"`.

Appendix E. Accessing code point boundaries

Mark Davis, IBM; Lauren Wood, SoftQuad Software Inc.

E.1. Introduction

This appendix is an informative, not a normative, part of the Level 3 DOM specification.

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

While indexing by code points as opposed to code units is not common in programs, some specifications such as [XPath 1.0] (and therefore XSLT and [XPointer]) use code point indices. For interfacing with such formats it is recommended that the programming language provide string processing methods for converting code point indices to code unit indices and back. Some languages do not provide these functions natively; for these it is recommended that the native `String` type that is bound to `DOMString` be extended to enable this conversion. An example of how such an API might look is supplied below.

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

E.2. Methods

Appendix F. IDL Definitions

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

The IDL files are also available as: http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/idl.zip

Appendix G. Java Language Binding

This appendix contains the complete Java [Java] bindings for the Level 3 Document Object Model Core.

The Java files are also available as http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/java-binding.zip

G.1. Java Binding Extension

☞ This section is informative.

This section defines the `DOMImplementationRegistry` object, discussed in § 1.3.7 – Bootstrapping on page 17, 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` interface.

package org.w3c.dom.bootstrap; import java.util.StringTokenizer; import java.util.Vector; import org.w3c.dom.DOMImplementationSource; import org.w3c.dom.DOMImplementationList; import org.w3c.dom.DOMImplementation; import java.io.InputStream; import java.io.BufferedReader; import java.io.InputStreamReader; import java.security.AccessController; import java.security.PrivilegedAction; /** * A factory that enables applications to obtain instances of * <code>DOMImplementation</code>. * * <p> * Example: * </p> * * <pre class='example'> * // get an instance of the DOMImplementation registry * DOMImplementationRegistry registry = * DOMImplementationRegistry.newInstance(); * // get a DOM implementation the Level 3 XML module * DOMImplementation domImpl = * registry.getDOMImplementation("XML 3.0"); * </pre> * * <p> * This provides an application with an implementation-independent starting * point. DOM implementations may modify this class to meet new security * standards or to provide *additional* fallbacks for the list of * DOMImplementationSources. * </p> * * @see DOMImplementation * @see DOMImplementationSource * @since DOM Level 3 */ public final class DOMImplementationRegistry { /** * The system property to specify the * DOMImplementationSource class names. */ public static final String PROPERTY = "org.w3c.dom.DOMImplementationSourceList"; /** * Default columns per line. */ private static final int DEFAULT_LINE_LENGTH = 80; /** * The list of DOMImplementationSources. */ private Vector sources; /** * Private constructor. * @param srcs Vector List of DOMImplementationSources */ private DOMImplementationRegistry(final Vector srcs) { sources = srcs; } /** * Obtain a new instance of a <code>DOMImplementationRegistry</code>. * * The <code>DOMImplementationRegistry</code> is initialized by the * application or the implementation, depending on the context, by * first checking the value of the Java system property * <code>org.w3c.dom.DOMImplementationSourceList</code> and * the the service provider whose contents are at * "<code>META_INF/services/org.w3c.dom.DOMImplementationSourceList</code>" * The value of this property is a white-space separated list of * names of availables classes implementing the * <code>DOMImplementationSource</code> interface. Each class listed * in the class name list is instantiated and any exceptions * encountered are thrown to the application. * * @return an initialized instance of DOMImplementationRegistry * @throws ClassNotFoundException * If any specified class can not be found * @throws InstantiationException * If any specified class is an interface or abstract class * @throws IllegalAccessException * If the default constructor of a specified class is not accessible * @throws ClassCastException * If any specified class does not implement * <code>DOMImplementationSource</code> */ public static DOMImplementationRegistry newInstance() throws ClassNotFoundException, InstantiationException, IllegalAccessException, ClassCastException { Vector sources = new Vector(); ClassLoader classLoader = getClassLoader(); // fetch system property: String p = getSystemProperty(PROPERTY); // // if property is not specified then use contents of // META_INF/org.w3c.dom.DOMImplementationSourceList from classpath if (p == null) { p = getServiceValue(classLoader); } if (p == null) { // // DOM Implementations can modify here to add *additional* fallback // mechanisms to access a list of default DOMImplementationSources. } if (p != null) { StringTokenizer st = new StringTokenizer(p); while (st.hasMoreTokens()) { String sourceName = st.nextToken(); // Use context class loader, falling back to Class.forName // if and only if this fails... Class sourceClass = null; if (classLoader != null) { sourceClass = classLoader.loadClass(sourceName); } else { sourceClass = Class.forName(sourceName); } DOMImplementationSource source = (DOMImplementationSource) sourceClass.newInstance(); sources.addElement(source); } } return new DOMImplementationRegistry(sources); } /** * Return the first implementation that has the desired * features, or <code>null</code> if none is found. * * @param features * A string that specifies which features are required. This is * a space separated list in which each feature is specified by * its name optionally followed by a space and a version number. * This is something like: "XML 1.0 Traversal +Events

2.0" * @return An implementation that has the desired features, * or <code>null</code> if none found. */ public DOMImplementation getDOMImplementation(final String features) { int size = sources.size(); String name = null; for (int i = 0; i < size; i++) { DOMImplementationSource source = (DOMImplementationSource) sources.elementAt(i); DOMImplementation impl = source.getDOMImplementation(features); if (impl != null) { return impl; } } return null; } /** * Return a list of implementations that support the * desired features. * * @param features * A string that specifies which features are required. This is * a space separated list in which each feature is specified by * its name optionally followed by a space and a version number. * This is something like: "XML 1.0 Traversal +Events 2.0" * @return A list of DOMImplementations that support the desired features. */ public DOMImplementationList getDOMImplementationList(final String features) { final Vector implementations = new Vector(); int size = sources.size(); for (int i = 0; i < size; i++) { DOMImplementationSource source = (DOMImplementationSource) sources.elementAt(i); DOMImplementationList impls = source.getDOMImplementationList(features); for (int j = 0; j < impls.getLength(); j++) { DOMImplementation impl = impls.item(j); implementations.addElement(impl); } } return new DOMImplementationList() { public DOMImplementation item(final int index) { if (index >= 0 && index < implementations.size()) { try { return (DOMImplementation) implementations.elementAt(index); } catch (ArrayIndexOutOfBoundsException e) { return null; } } return null; } public int getLength() { return implementations.size(); } }; } /** * Register an implementation. * * @param s The source to be registered, may not be <code>null</code> */ public void addSource(final DOMImplementationSource s) { if (s == null) { throw new NullPointerException(); } if (!sources.contains(s)) { sources.addElement(s); } } /** * * Gets a class loader. * * @return A class loader, possibly <code>null</code> */ private static ClassLoader getClassLoader() { try { ClassLoader contextClassLoader = getContextClassLoader(); if (contextClassLoader != null) { return contextClassLoader; } } catch (Exception e) { // Assume that the DOM application is in a JRE 1.1, use the // current ClassLoader return DOMImplementationRegistry.class.getClassLoader(); } return DOMImplementationRegistry.class.getClassLoader(); } /** * This method attempts to return the first line of the resource * META_INF/services/org.w3c.dom.DOMImplementationSourceList * from the provided ClassLoader. * * @param classLoader classLoader, may not be <code>null</code>. * @return first line of resource, or <code>null</code> */ private static String getServiceValue(final ClassLoader classLoader) { String serviceId = "META-INF/services/" + PROPERTY; // try to find services in CLASSPATH try { InputStream is = getResourceAsStream(classLoader, serviceId); if (is != null) { BufferedReader rd; try { rd = new BufferedReader(new InputStreamReader(is, "UTF-8"), DEFAULT_LINE_LENGTH); } catch (java.io.UnsupportedEncodingException e) { rd = new BufferedReader(new InputStreamReader(is), DEFAULT_LINE_LENGTH); } String serviceValue = rd.readLine(); rd.close(); if (serviceValue != null && serviceValue.length() > 0) { return serviceValue; } } } catch (Exception ex) { return null; } return null; } /** * A simple JRE (Java Runtime Environment) 1.1 test * * @return <code>true</code> if JRE 1.1 */ private static boolean isJRE11() { try { Class c = Class.forName("java.security.AccessController"); // java.security.AccessController existed since 1.2 so, if no // exception was thrown, the DOM application is running in a JRE // 1.2 or higher return false; } catch (Exception ex) { // ignore } return true; } /** * This method returns the ContextClassLoader or <code>null</code> if * running in a JRE 1.1 * * @return The Context Classloader */ private static ClassLoader getContextClassLoader() { return isJRE11() ? null : (ClassLoader) AccessController.doPrivileged(new PrivilegedAction() { public Object run() { ClassLoader classLoader = null; try { classLoader = Thread.currentThread().getContextClassLoader(); } catch (SecurityException ex) { } return classLoader; } }); } /** * This method returns the system property indicated by the specified name * after checking access control privileges. For a JRE 1.1, this check is * not done. * * @param name the name of the system property * @return the system property */ private static String getSystemProperty(final String name) { return isJRE11() ? (String) System.getProperty(name) : (String) AccessController.doPrivileged(new PrivilegedAction() { public Object run() { return System.getProperty(name); } }); } /** * This method returns an Inputstream for the reading resource * META_INF/services/org.w3c.dom.DOMImplementationSourceList after checking * access control privileges. For a JRE

1.1, this check is not done. * * @param classLoader classLoader * @param name the resource * @return an Inputstream for the resource specified */ private static InputStream getResourceAsStream(final ClassLoader classLoader, final String name) { if (isJRE11()) { InputStream ris; if (classLoader == null) { ris = ClassLoader.getSystemResourceAsStream(name); } else { ris = classLoader.getResourceAsStream(name); } return ris; } else { return (InputStream) AccessController.doPrivileged(new PrivilegedAction() { public Object run() { InputStream ris; if (classLoader == null) { ris = ClassLoader.getSystemResourceAsStream(name); } else { ris = classLoader.getResourceAsStream(name); } return ris; } }); } } }

G.2. Other Core interfaces

Appendix H. ECMAScript Language Binding

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

H.1. ECMAScript Binding Extension

This section defines the `DOMImplementationRegistry` object, discussed in § 1.3.7 – Bootstrapping on page 17, for ECMAScript.

Objects that implements the DOMImplementationRegistry interface

DOMImplementationRegistry is a global variable which has the following functions:

getDOMImplementation(features)

This method returns the first registered object that implements the DOMImplementation interface and has the desired features, or null if none is found.

The features parameter is a String. See also `DOMImplementationSource.getDOMImplementation()`.

getDOMImplementationList(features)

This method returns a `DOMImplementationList` list of registered object that implements the DOMImplementation interface and has the desired features.

The features parameter is a String. See also `DOMImplementationSource.getDOMImplementationList()`.

H.2. Other Core interfaces

☞ In addition of having `DOMConfiguration` parameters exposed to the application using the `setParameter` and `getParameter`, those parameters are also exposed as ECMAScript properties on the `DOMConfiguration` object. The name of the parameter is converted into a property name using a camel-case convention: the character '-' (HYPHEN-MINUS) is removed and the following character is being replaced by its uppercase equivalent.

Appendix I. Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including participants of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Andrew Watson (Object Management Group), Andy Heninger (IBM), Angel Diaz (IBM), Arnaud Le Hors (W3C and IBM), Ashok Malhotra (IBM and Microsoft), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Ezell (Hewlett-Packard Company), David Singer (IBM), Dimitris Dimitriadis (Improve AB and invited expert), Don Park (invited), Elena Litani (IBM), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Jeroen van Rotterdam (X-Hive Corporation), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape/AOL), Jon Ferraiolo (Adobe), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Lauren Wood (SoftQuad Software Inc., *former Chair*), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mary Brady (NIST), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégaret (W3C, *W3C Team Contact and former Chair*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL, *Chair*), Rezaur Rahman (Intel), Rich Rollman (Microsoft), Rick Gessner (Netscape), Rick Jelliffe (invited), Rob Relyea (Microsoft), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tim Yu (Oracle), Tom Pixley (Netscape/AOL), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections (Please, keep bugging us with your issues!).

Many thanks to Andrew Clover, Petteri Stenius, Curt Arnold, Glenn A. Adams, Christopher Aillon, Scott Nichol, François Yergeau, Anjana Manian, Susan Lesch, and Jeffery B. Rancier for their review and comments of this document.

Special thanks to the DOM Conformance Test Suites contributors: Fred Drake, Mary Brady (NIST), Rick Rivello (NIST), Robert Clary (Netscape), with a special mention to Curt Arnold.

I.1. Production Systems

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

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

After DOM Level 1, we used Xerces as the basis DOM implementation and wish to thank the authors. Philippe Le Hégaret and Arnaud Le Hors wrote the Java programs which are the DOM application.

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

Appendix J. Glossary

Arnaud Le Hors, W3C; Robert S. Sutor, IBM Research (for DOM Level 1)
Some of the following term definitions have been borrowed or modified from similar definitions in other W3C or standards documents. See the links within the definitions for more information.

16-bit unit

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

ancestor

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

API

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

anonymous type name

An *anonymous type name* is an implementation-defined, globally unique qualified name provided by the processor for every anonymous type declared in a .

bubbling phase

The process by which an can be handled by one of the target ancestors after being handled by the .

capture phase

The process by which an can be handled by one of the target ancestors before being handled by the .

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.

content model

The *content model* is a simple grammar governing the allowed types of the child elements and the order in which they appear. See Element Content in XML [XML 1.0].

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*.

cooked model

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

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

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

CORBA

CORBA is the *Common Object Request Broker Architecture* from the OMG [CORBA]. This architecture is a collection of objects and libraries that allow the creation of applications containing objects that make and receive requests and responses in a distributed environment.

cursor

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

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.

deprecation

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

descendant

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

document element

There is only one document element in a `Document`. 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 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.

DOM Level 0

The term "DOM Level 0" refers to a mix (not formally specified) of HTML document functionalities offered by Netscape Navigator version 3.0 and Microsoft Internet Explorer version 3.0. In some cases, attributes or methods have been included for reasons of backward compatibility with "DOM Level 0".

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].

event

An event is the representation of some asynchronous occurrence (such as a mouse click on the presentation of the element, or the removal of child node from an element, or any of unthinkably many other possibilities) that gets associated with an .

event target

The object to which an is targeted.

equivalence

Two nodes are *equivalent* if they have the same node type and same node name. Also, if the nodes contain data, that must be the same. Finally, if the nodes have attributes then collection of attribute names must be the same and the attributes corresponding by name must be equivalent as nodes.

Two nodes are *deeply equivalent* if they are *equivalent*, the child node lists are equivalent are equivalent as `NodeList` objects, and the pairs of equivalent attributes must in fact be deeply equivalent.

Two `NodeList` objects are *equivalent* if they have the same length, and the nodes corresponding by index are deeply equivalent.

Two `NamedNodeMap` objects are *equivalent* if they have the same length, they have same collection of names, and the nodes corresponding by name in the maps are deeply equivalent.

Two `DocumentType` nodes are *equivalent* if they are equivalent as nodes, have the same names, and have equivalent entities and attributes `NamedNodeMap` objects.

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` or `CDATASection` nodes that can be visited sequentially in or in reversed document order without entering, exiting, or passing over `Element`, `Comment`, or `ProcessingInstruction` nodes.

global declaration

A *global declaration* is a schema declaration, usually for an element or an attribute, that is available for use in content models throughout the , i.e. a declaration that is not bound to a particular context.

hosting implementation

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

HTML

The HyperText Markup Language (*HTML*) is a simple markup language used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of applications. [HTML 4.01]

IDL

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

implementor

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

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.

initial structure model

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

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.

live

An object is *live* if any change to the underlying document structure is reflected in the object.

local name

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

method

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

model

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

namespace prefix

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

namespace URI

A *namespace URI* is a URI that identifies an XML namespace. This is called the namespace name in Namespaces in XML [XML Namespaces]. See also sections 1.3.2 "DOM URIs" and 1.3.3 "XML Namespaces" regarding URIs and namespace URIs handling and comparison in the DOM APIs.

namespace well-formed

A node is a *namespace well-formed* XML node if it is a node, and follows the productions and namespace constraints. If [XML 1.0] is used, the constraints are defined in [XML Namespaces]. If [XML 1.1] is used, the constraints are defined in [XML Namespaces 1.1].

object model

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

parent

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

partially valid

A node in a DOM tree is *partially valid* if it is (this part is for comments and processing instructions) and its immediate children are those expected by the content model. The node may be missing trailing required children yet still be considered *partially valid*.

qualified name

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

read only node

A *read only node* is a node that is immutable. This means its list of children, its content, and its attributes, when it is an element, cannot be changed in any way. However, a read only node can possibly be moved, when it is not itself contained in a read only node.

root node

The *root node* is a node that is not a child of any other node. All other nodes are children or other descendants of the root node.

schema

A *schema* defines a set of structural and value constraints applicable to XML documents. Schemas can be expressed in schema languages, such as DTD, XML Schema, etc.

sibling

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

string comparison

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

tag valid document

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

target node

The target node is the node representing the to which an is targeted using the DOM event flow.

target phase

The process by which an can be handled by the .

token

An information item such as an XML Name which has been .

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.

type valid document

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

uncooked model

See initial structure model.

well-formed

A node is a *well-formed* XML node if its serialized form, without doing any transformation during its serialization, matches its respective production in [XML 1.0] or [XML 1.1] (depending on the XML version in use) with all well-formedness constraints related to that production, and if the entities which are referenced within the node are also well-formed. If namespaces for XML are in use, the node must also be .

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 [IETF RFC 2396], which are used in XML documents as element types and attribute names. [XML Namespaces]

Appendix K. References

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

Canonical XML

Canonical XML Version 1.0, J. Boyer, Editor. World Wide Web Consortium, 15 March 2001. This version of the Canonical XML Recommendation is http://www.w3.org/TR/2001/REC-xml-c14n-20010315. The latest version of Canonical XML is available at http://www.w3.org/TR/xml-c14n. Available at `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`.

CharModel

Character Model for the World Wide Web 1.0: Normalization, M. Dürst, F. Yergeau, R. Ishida, M. Wolf, T. Texin, and A. Phillips, Editors. World Wide Web Consortium, February 2004. This version of the Character Model for the World Wide Web 1.0: Normalization specification is http://www.w3.org/TR/2004/WD-charmod-norm-20040225. The latest version of Character Model for the World Wide Web 1.0: Normalization is available at http://www.w3.org/TR/charmod-norm. Available at `http://www.w3.org/TR/2004/WD-charmod-norm-20040225`.

COM

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

CORBA

The Common Object Request Broker: Architecture and Specification, version 2. Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm. Available at `http://www.omg.org/technology/documents/formal/corba_2.htm`.

CSS2

Cascading Style Sheets, level 2 Specification, B. Bos, H. Wium Lie, C. Lilley, and I. Jacobs, Editors. World Wide Web Consortium, 12 May 1998. This version of the Cascading Style Sheets Recommendation is http://www.w3.org/TR/1998/REC-CSS2-19980512. The latest version of Cascading Style Sheets is available at http://www.w3.org/TR/REC-CSS2. Available at `http://www.w3.org/TR/1998/REC-CSS2-19980512`.

DOM Level 1

DOM Level 1 Specification, V. Apparao, et al., Editors. World Wide Web Consortium, 1 October 1998. This version of the DOM Level 1 Recommendation is http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001. The latest version of DOM Level 1 is available at http://www.w3.org/TR/REC-DOM-Level-1. Available at `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`.

DOM Level 2 Core

Document Object Model Level 2 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, 13 November 2000. This version of the DOM Level 2 Core Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113. The latest version of DOM Level 2 Core is available at http://www.w3.org/TR/DOM-Level-2-Core. Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`.

DOM Level 3 Core

Document Object Model Level 3 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, 7 April 2004. This version of the Document Object Model Level 3 Core Recommendation is http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407. The latest version of DOM Level 3 Core is available at http://www.w3.org/TR/DOM-Level-3-Core. Available at `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`.

DOM Level 2 Events

Document Object Model Level 2 Events Specification, T. Pixley, Editor. World Wide Web Consortium, 13 November 2000. This version of the Document Object Model Level 2 Events Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113. The latest version of Document Object Model Level 2 Events is available at http://www.w3.org/TR/DOM-Level-2-Events. Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113`.

DOM Level 3 Events

Document Object Model Level 3 Events Specification, P. Le Hégaret, T. Pixley, Editors. World Wide Web Consortium, November 2003. This version of the Document Object Model Level 3 Events specification is http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107. The latest version of Document Object Model Level 3 Events is available at http://www.w3.org/TR/DOM-Level-3-Events. Available at `http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107`.

DOM Level 3 Load and Save

Document Object Model Level 3 Load and Save Specification, J. Stenback, A. Heninger, Editors. World Wide Web Consortium, 7 April 2004. This version of the DOM Level 3 Load and Save Recommendation is http://www.w3.org/TR/2004/REC-DOM-Level-3-LS-20040407. The latest version of DOM Level 3 Load and Save is available at http://www.w3.org/TR/DOM-Level-3-LS. Available at `http://www.w3.org/TR/2004/REC-DOM-Level-3-LS-20040407`.

DOM Level 2 HTML

Document Object Model Level 2 HTML Specification, J. Stenback, et al., Editors. World Wide Web Consortium, 9 January 2003. This version of the Document Object Model Level 2 HTML Recommendation is http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109. The latest version of Document Object Model Level 2 HTML is available at http://www.w3.org/TR/DOM-Level-2-HTML. Available at `http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109`.

DOM Level 3 Requirements

"DOM Requirements for DOM Level 3" in *DOM Requirements for DOM Level 3*, B. Chang, et al., Editors. World Wide Web Consortium, February 2004. This version of the DOM Requirements for DOM Level 3 is http://www.w3.org/TR/2004/NOTE-DOM-Requirements-20040226#Level3. The latest version of DOM Requirements is available at http://www.w3.org/TR/DOM-Requirements. Available at `http://www.w3.org/TR/2004/NOTE-DOM-Requirements-20040226#Level3`.

DOM Level 2 Style Sheets and CSS

Document Object Model Level 2 Style Sheets and CSS Specification, C. Wilson, P. Le Hégaret, V. Apparao, Editors. World Wide Web Consortium, 13 November 2000. This version of the Document Object Model Level 2 Style Sheets and CSS Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113. The latest version of Document Object Model Level 2 Style Sheets and CSS is available at http://www.w3.org/TR/DOM-Level-2-Style. Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113`.

DOM Level 2 Traversal and Range

Document Object Model Level 2 Traversal and Range Specification, J. Kesselman, J. Robie, M. Champion, P. Sharpe, V. Apparao, L. Wood, Editors. World Wide Web Consortium, 13 November 2000. This version of the Document Object Model Level 2 Traversal and Range Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113. The latest version of Document Object Model Level 2 Traversal and Range is available at http://www.w3.org/TR/DOM-Level-2-Traversal-Range. Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113`.

DOM Level 3 Validation

Document Object Model Level 3 Validation Specification, B. Chang, J. Kesselman, R. Rahman, Editors. World Wide Web Consortium, 27 January 2003. This version of the DOM Level 3 Validation Recommendation is http://www.w3.org/TR/2004/REC-DOM-Level-3-Val-20040127/. The latest version of DOM Level 3 Validation is available at http://www.w3.org/TR/DOM-Level-3-Val. Available at `http://www.w3.org/TR/2004/REC-DOM-Level-3-Val-20040127/`.

DOM Level 2 Views

Document Object Model Level 2 Views Specification, A. Le Hors, L. Cable, Editors. World Wide Web Consortium, 13 November 2000. This version of the Document Object Model Level 2 Views Recommendation is http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113. The latest version of Document Object Model Level 2 Views is available at http://www.w3.org/TR/DOM-Level-2-Views. Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113`.

DOM Level 3 XPath

Document Object Model Level 3 XPath Specification, R. Whitmer, Editor. World Wide Web Consortium, March 2003. This version of the Document Object Model Level 3 XPath specification is http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226. The latest version of Document Object Model Level 3 XPath is available at http://www.w3.org/TR/DOM-Level-3-XPath. Available at `http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226/`.

DWW95

Developing International Software for Windows 95 and Windows NT: A Handbook for International Software Design, N. Kano, Author. Microsoft Press, 1995. ISBN 1-55615-840-8.

ECMAScript

ECMAScript Language Specification, Third Edition. European Computer Manufacturers Association, Standard ECMA-262, December 1999. This version of the ECMAScript Language is available from http://www.ecma-international.org/.

HTML 4.01

HTML 4.01 Specification, D. Raggett, A. Le Hors, and I. Jacobs, Editors. World Wide Web Consortium, 17 December 1997, revised 24 April 1998, revised 24 December 1999. This version of the HTML 4.01 Recommendation is http://www.w3.org/TR/1999/REC-html401-19991224. The latest version of HTML 4 is available at http://www.w3.org/TR/html4. Available at `http://www.w3.org/TR/1999/REC-html401-19991224/`.

IANA-CHARSETS

Official Names for Character Sets, K. Simonsen, et al., Editors. Internet Assigned Numbers Authority. Available at ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets. Available at `ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets`.

ISO/IEC 10646

ISO/IEC 10646-2000 (E). Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization, 2000. See also International Organization for Standardization, available at http://www.iso.ch, for the latest version.

Java

The Java Language Specification, J. Gosling, B. Joy, and G. Steele, Authors. Addison-Wesley, September 1996. Available at http://java.sun.com/docs/books/jls Available at `http://java.sun.com/docs/books/jls`.

Java IDL

Java IDL. Sun Microsystems. Available at http://java.sun.com/products/jdk/idl/ Available at `http://java.sun.com/products/jdk/idl/`.

JavaScript

JavaScript Resources. Netscape Communications Corporation. Available at http://devedge.netscape.com/central/javascript/ Available at `http://devedge.netscape.com/central/javascript/`.

JAXP

Java API for XML Processing (JAXP). Sun Microsystems. Available at http://java.sun.com/xml/jaxp/. Available at `http://java.sun.com/xml/jaxp/`.

JScript

JScript Resources. Microsoft. Available at http://msdn.microsoft.com/library/en-us/script56/html/js56jslrfjscriptlanguagereference.asp Available at `http://msdn.microsoft.com/library/en-us/script56/html/js56jslr-fjscriptlanguagereference.asp`.

KeyEvent for Java

Java 2 Platform, Standard Edition, v. 1.4.2 API Specification, Class java.awt.events.KeyEvent. Sun Microsystems. Available at http://java.sun.com/j2se/1.4.2/docs/api/java/awt/event/KeyEvent.html. Available at `http://java.sun.com/j2se/1.4.2/docs/api/java/awt/event/KeyEvent.html`.

Keys enumeration for .Net

.NET Framework Class Library, Keys Enumeration. Microsoft. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWindows-FormsKeysClassTopic.asp. Available at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWindowsFormsKeysClassTopic.asp`.

MathML 2.0

Mathematical Markup Language (MathML) Version 2.0 (Second Edition), D. Carlisle, P. Ion, R. Miner, N. Poppelier, Editors. World Wide Web Consortium, 21 October 2001, revised 21 February 2001. This version of the Math 2.0 Recommendation is http://www.w3.org/TR/2003/REC-MathML2-20031021. The latest version of MathML 2.0 is available at http://www.w3.org/TR/MathML2. Available at `http://www.w3.org/TR/2003/REC-MathML2-20031021`.

MIDL

MIDL Language Reference. Microsoft. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_language_reference.asp. Available at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_language_reference.asp`.

OMG IDL

"OMG IDL Syntax and Semantics" defined in *The Common Object Request Broker: Architecture and Specification, version 2*, Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm. Available at `http://www.omg.org/technology/documents/formal/corba_2.htm`.

IETF IRIs

Internationalized Resource Identifiers (IRIs), M. Dürst, and M. Suignard, Authors. Internet Engineering Task Force, March 2003. Available at http://www.w3.org/International/iri-edit/draft-duerst-iri-03.txt. Available at `http://www.w3.org/International/iri-edit/draft-duerst-iri-03.txt`.

IETF RFC 2396

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

IETF RFC 2616

Hypertext Transfer Protocol -- HTTP/1.1, R. Fielding, et al., Authors. Internet Engineering Task Force, June 1999. Available at http://www.ietf.org/rfc/rfc2616.txt. Available at `http://www.ietf.org/rfc/rfc2616.txt`.

IETF RFC 3023

XML Media Types, M. Murata, S. St.Laurent, and D. Kohn, Editors. Internet Engineering Task Force, January 2001. Available at http://www.ietf.org/rfc/rfc3023.txt. Available at `http://www.ietf.org/rfc/rfc3023.txt`.

SAX

Simple API for XML, D. Megginson and D. Brownell, Maintainers. Available at http://www.sax-project.org/. Available at `http://www.saxproject.org/`.

SVG 1.1

Scalable Vector Graphics (SVG) 1.1 Specification, J. Ferraiolo, (FUJISAWA Jun), and D. Jackson, Editors. World Wide Web Consortium, 14 January 2003. This version of the SVG 1.1 Recommendation is http://www.w3.org/TR/2003/REC-SVG11-20030114/. The latest version of SVG 1.1 is available at http://www.w3.org/TR/SVG. Available at `http://www.w3.org/TR/2003/REC-SVG11-20030114/`.

Unicode

The Unicode Standard, Version 4, ISBN 0-321-18578-1, as updated from time to time by the publication of new versions. The Unicode Consortium, 2000. See also Versions of the Unicode Standard, available at http://www.unicode.org/unicode/standard/versions, for latest version and additional information on versions of the standard and of the Unicode Character Database.

UTR #15

Unicode Normalization Forms, The Unicode Standard Annex #15. The Unicode Consortium, 2003. The latest version of this annex is available at http://www.unicode.org/unicode/reports/tr15/. Available at `http://www.unicode.org/unicode/reports/tr15/`.

VoiceXML 2.0

Voice Extensible Markup Language (VoiceXML) Version 2.0, S. McGlashan, et al., Editors. World Wide Web Consortium, 16 March 2004. This version of the Voice Extensible Markup Language Version 2.0 Recommendation is http://www.w3.org/TR/2004/REC-voicexml20-20040316. The latest version of Voice Extensible Markup Language Version 2.0 is available at http://www.w3.org/TR/voicexml20/. Available at `http://www.w3.org/TR/2004/REC-voicexml20-20040316`.

XForms 1.0

XForms 1.0, M. Dubinko, et al., Editors. World Wide Web Consortium, 14 October 2003. This version of the XForms 1.0 Recommendation is http://www.w3.org/TR/2003/REC-xforms-20031014/. The latest version of XForms 1.0 is available at http://www.w3.org/TR/xforms/. Available at `http://www.w3.org/TR/2003/REC-xforms-20031014/`.

XHTML 1.0

XHTML 1.0: The Extensible HyperText Markup Language, S. Pemberton, et al., Authors. World Wide Web Consortium, 26 January 2000, revised 1 August 2002. This version of the XHTML 1.0 Recommendation is http://www.w3.org/TR/2002/REC-xhtml1-20020801. The latest version of XHTML 1.0 is available at http://www.w3.org/TR/xhtml1. Available at `http://www.w3.org/TR/2002/REC-xhtml1-20020801`.

XQuery 1.0 and XPath 2.0 Data Model

W3C (World Wide Web Consortium) XQuery 1.0 and XML Path 2.0 Data Model, November 2003. Available at http://www.w3.org/TR/query-datamodel

XML 1.0

Extensible Markup Language (XML) 1.0 (Third Edition), T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Editors. World Wide Web Consortium, 4 February 2004, revised 10 February 1998 and 6 October 2000. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2004/REC-xml-20040204. The latest version of XML 1.0 is available at http://www.w3.org/TR/REC-xml. Available at `http://www.w3.org/TR/2004/REC-xml-20040204`.

XML 1.1

XML 1.1, T. Bray, and al., Editors. World Wide Web Consortium, 4 February 2004. This version of the XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml11-20040204. The latest version of XML 1.1 is available at http://www.w3.org/TR/xml11. Available at `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

XML Base

XML Base, J. Marsh, Editor. World Wide Web Consortium, June 2001. This version of the XML Base Recommendation is http://www.w3.org/TR/2001/REC-xmlbase-20010627. The latest version of XML Base is available at http://www.w3.org/TR/xmlbase. Available at `http://www.w3.org/TR/2001/REC-xmlbase-20010627/`.

XML Information Set

XML Information Set (Second Edition), J. Cowan and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004, revised 24 October 2001. This version of the XML Information Set Recommendation is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version of XML Information Set is available at http://www.w3.org/TR/xml-infoset. Available at `http://www.w3.org/TR/2004/REC-xml-infoset-20040204/`.

XML Events

XML Events, S. McCarron, S. Pemberton, and T.V. Raman, Editors. World Wide Web Consortium, August 2003. This version of the XML Events specification is http://www.w3.org/TR/2003/PR-xml-events-20030804. The latest version of XML Events is available at http://www.w3.org/TR/xml-events. Available at `http://www.w3.org/TR/2003/PR-xml-events-20030804/`.

XML Namespaces

Namespaces in XML, T. Bray, D. Hollander, and A. Layman, Editors. World Wide Web Consortium, 14 January 1999. This version of the Namespaces in XML Recommendation is http://www.w3.org/TR/1999/REC-xml-names-19990114. The latest version of Namespaces in XML is available at http://www.w3.org/TR/REC-xml-names. Available at `http://www.w3.org/TR/1999/REC-xml-names-19990114/`.

XML Namespaces 1.1

Namespaces in XML 1.1, T. Bray, D. Hollander, A. Layman, and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004. This version of the Namespaces in XML 1.1 Recommendation is http://www.w3.org/TR/2004/REC-xml-names11-20040204. The latest version of Namespaces in XML 1.1 is available at http://www.w3.org/TR/xml-names11/. Available at `http://www.w3.org/TR/2004/REC-xml-names11-20040204/`.

XML Schema Part 0

XML Schema Part 0: Primer, D. Fallside, Editor. World Wide Web Consortium, 2 May 2001. This version of the XML Part 0 Recommendation is http://www.w3.org/TR/2001/REC-xmlschema-0-20010502. The latest version of XML Schema Part 0 is available at http://www.w3.org/TR/xmlschema-0. Available at `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`.

XML Schema Part 1

XML Schema Part 1: Structures, H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Part 1 Recommendation is http://www.w3.org/TR/2001/REC-xmlschema-1-20010502. The latest version of XML Schema Part 1 is available at http://www.w3.org/TR/xmlschema-1. Available at `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`.

XML Schema Part 2

XML Schema Part 2: Datatypes, P. Byron and Ashok Malhotra, Editors. World Wide Web Consortium, 2 May 2001. This version of the XML Part 2 Recommendation is http://www.w3.org/TR/2001/REC-xmlschema-2-20010502. The latest version of XML Schema Part 2 is available at http://www.w3.org/TR/xmlschema-2. Available at `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

XML StyleSheet

W3C (World Wide Web Consortium) Associating Style Sheets with XML documents Version 1.0, June 1999. Available at http://www.w3.org/1999/06/REC-xml-stylesheet-19990629

XPath 1.0

XML Path Language (XPath) Version 1.0, J. Clark and S. DeRose, Editors. World Wide Web Consortium, 16 November 1999. This version of the XPath 1.0 Recommendation is http://www.w3.org/TR/1999/REC-xpath-19991116. The latest version of XPath 1.0 is available at http://www.w3.org/TR/xpath. Available at `http://www.w3.org/TR/1999/REC-xpath-19991116`.

XPointer

XPointer Framework, P. Grosso, E. Maler, J. Marsh, and N. Walsh., Editors. World Wide Web Consortium, 25 March 2003. This version of the XPointer Framework Recommendation is http://www.w3.org/TR/2003/REC-xptr-framework-20030325/. The latest version of XPointer Framework is available at http://www.w3.org/TR/xptr-framework/. Available at `http://www.w3.org/TR/2003/REC-xptr-framework-20030325/`.

Appendix L. Index

This page is intentionally left blank.