previous next contents index

09 April 2002

What is the Document Object Model?

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

Introduction

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

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

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

IDL specification, we provide *language bindings* for Java [Java] and ECMAScript [ECMAScript] (an industry-standard scripting language based on JavaScript [JavaScript] and JScript [JScript]).

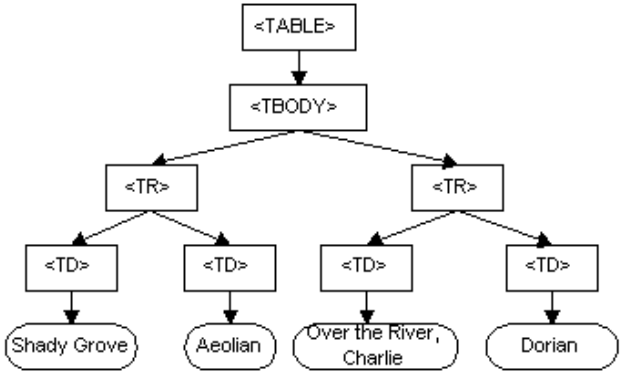
Note: OMG IDL is used only as a language-independent and implementation-neutral way to specify *interfaces*. Various other IDLs could have been used ([COM], [Java IDL], [MIDL], ...). In general, IDLs are designed for specific computing environments. The Document Object Model can be implemented in any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

What the Document Object Model is

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

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

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



graphical representation of the DOM of the example table

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

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

The name "Document Object Model" was chosen because it is an "*object model*" in the traditional object oriented design sense: documents are modeled using objects, and the model encompasses not only the structure of a document, but also the behavior of a document and the objects of which it is composed. In other

words, the nodes in the above diagram do not represent a data structure, they represent objects, which have functions and identity. As an object model, the DOM identifies:

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

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

What the Document Object Model is not

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

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

implemented using language-independent systems like COM or CORBA; it may also be implemented using language-specific bindings like the Java or ECMAScript bindings specified in this document.

Where the Document Object Model came from

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

Entities and the DOM Core

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

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

the "&" will be replaced by the character "&", and the text in the P element will form a single continuous sequence of characters. Since numeric character references and pre-defined entities are not recognized as such in CDATA sections, or in the SCRIPT and STYLE elements in HTML, they are not replaced by the single character they appear to refer to. If the example above were enclosed in a CDATA section, the "&" would not be replaced by "&"; neither would the <p> be recognized as a start tag. The representation of general entities, both internal and external, are defined within the extended (XML) interfaces of Document Object Model Core.

Note: When a DOM representation of a document is serialized as XML or HTML text, applications will need to check each character in text data to see if it needs to be escaped using a numeric or pre-defined entity. Failing to do so could result in invalid HTML or XML. Also, *implementations* should be aware of the fact that serialization into a character encoding ("charset") that does not fully cover ISO 10646 may fail if there are characters in markup or CDATA sections that are not present in the encoding.

Conformance

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

An implementation is DOM Level 3 conformant if it supports the Core module defined in this document (see Fundamental Interfaces). 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].
Mutation Events module
defines the feature *"MutationEvents"* in [DOM Level 3 Events].
HTML Events module
defines the feature *"HTMLEvents"* in [DOM Level 3 Events].

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

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

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.

---

previous next contents index