ptc

arbortext®

Programmer's Reference

8.1.2.0

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

Contents

About This Guide

This guide covers the following information:

- Part 1: Getting Started — Introduces the AOM and describes supported program and script languages.
- Part 2: Using the AOM — Describes configuration and customizations necessary to implement custom applications and how to use Java, JavaScript, JScript, VBScript, COM, and C++ to access the AOM.
- Part 3: Programming and scripting techniques — Provides descriptions and examples of using Arbortext Editor and the AOM to perform basic document operations and to work with events.
- Part 4: Interfaces — Details the W3C and Arbortext interfaces (and their attributes, enumerations, and methods) supported by the AOM and the Arbortext Publishing Engine.

Prerequisite Knowledge

The *Programmer's Reference* assumes advanced skill using Java, JavaScript, JScript, VBScript, or COM (Component Object Model). If you're creating a Arbortext Publishing Engine application, you also need to be familiar with Java servlets, servlet containers, web servers, the HTTP protocol, and the SOAP protocol.

Documentation for PTC Products

You can access PTC product documentation using the following resources:

- Online Help

Click **Help** from the user interface for online help available for the product.

- Reference Documentation

  PDFs of reference information are available from the Product Documentation area of www.ptc.com/support.

  Select the Arbortext tab to access the Arbortext Reference Documentation link.

- Help Center

  Help Centers for the most recent product releases are available from the Product Documentation area of www.ptc.com/support.

  Select the Arbortext tab to access the Help Centers link.

You must have a Service Contract Number (SCN) before you can access the Arbortext Reference Documentation or Help Centers links. If you do not have an SCN, contact PTC Technical Support or Customer Care Departments using the contact instructions found in your Customer Support Guide.

Global Services

PTC Global Services delivers the highest quality, most efficient and most comprehensive deployments of the PTC Product Development System including Creo, Windchill, Arbortext, and PTC Mathcad. PTC's Implementation and Expansion solutions integrate the process consulting, technology implementation, education and value management activities customers need to be successful. Customers are led through Solution Design, Solution Development and Solution Deployment phases with the continuous driving objective of maximizing value from their investment.

Contact your PTC sales representative for more information on Global Services.

Comments

PTC welcomes your suggestions and comments on our documentation. You can submit your feedback to the following email address:

arbortext-documentation@ptc.com

Please include the following information in your email:

- Name
- Company
- Product
- Product Release
- Document or Online Help Topic Title

- Level of Expertise in the Product (Beginning, Intermediate, Advanced)
- Comments (including page numbers where applicable)

Documentation Conventions

This guide uses the following notational conventions:

- **Bold text** represents exact text that appears in the program's user interface. This includes items such as button text, menu selections, and dialog box elements. For example,

  Click **OK** to begin the operation.
- A right arrow represents successive menu selections. For example,

  Choose **File ► Print** to print the document.
- `Monospaced text` represents code, command names, file paths, or other text that you would type exactly as described. For example,

  At the command line, type `version` to display version information.
- *`Italicized monospaced text`* represents variable text that you would type. For example,

  `installation-dir``\custom\scripts\`
- *Italicized text* represents a reference to other published material. For example,

  If you are new to the product, refer to the *Getting Started Guide* for basic interface information.

Conventions Used in This Guide

In addition to the conventions listed earlier, this guide uses the following notational conventions:

- Square braces (**[ ]**) denote optional parameters which may be omitted. For example:

  insertBefore(**newChild**[, **refChild**])
- A vertical bar (**|**) separates parameters in a list from which one parameter must be chosen or used. For example:

  allowinvalidmarkup {**on** | **off**}

List of Terms

The following terms are used throughout this guide.

- **AOM** — Arbortext Object Model.
- **attributes** — [Definition TBD]
- **interfaces** — [Definition TBD]
- **methods** — [Definition TBD]
- **multicell** — A rectangular array of spanned cells in a table.
- **OID** — [Definition TBD]
- **properties** — [Definition TBD]
- **scripts** — [Definition TBD]
- **TOID** — Table Object Identifier.

Where to Get More Information

Supporting documentation and related Javadoc for Arbortext Editor and Arbortext Publishing Engine can be found in the Arbortext Editor Help Center. You can open the Help Center from the Arbortext Editor **Help** menu. ACL (Arbortext Command Language) documentation is included in the Help Center and is not the focus of the *Programmer's Reference*.

If you're using the Arbortext Publishing Engine, be sure to review *Installation Guide for Arbortext Publishing Engine* and *Configuration Guide for Arbortext Publishing Engine* for extensive information on Arbortext Publishing Engine installation, setup, and configuration.

Training classes are also available. For more information, visit www.ptc.com.

If you are looking for more general information on programming or scripting languages, you may want to consult the following resources:

- *Thinking in Java*, by Bruce Eckel. Published by Prentice Hall PTR.
- Oracle has extensive Java information available at its web site www.oracle.com/technetwork/java/index.html. The tutorials are especially helpful to beginners.
- *JavaScript: The Definitive Guide*, by David Flanagan. Published by O'Reilly and Associates Inc.
- Mozilla has extensive JavaScript information available at its web site www.mozilla.org.
- ECMA International (European Computer Manufacturers Association) has the *ECMAScript Language Specification*, which is the standard used for JavaScript, available at its web site www.ecma.ch.
- Microsoft has extensive information about JScript, VBScript, ActiveX scripting host, and COM available at its web site msdn.microsoft.com.

Getting Started

1

Supported Program and Script Languages

You can write programs and scripts in several supported languages. The following table lists the supported languages and their descriptions:

Supported Program and Script Languages

Language	Description
Java	Cross-platform, object-oriented programming language.
COM	Windows Component Object Model. COM is not actually a language but a standard. It is supported by several languages, including C++ and Visual Basic.
JavaScript	Cross-platform, object-oriented scripting language, not directly related to Java. The standard it follows is called ECMAScript.
JScript	A COM-based, loosely-typed scripting language, not directly related to Java but similar to JavaScript.
VBScript	A COM-based scripting language that is a subset of the Visual Basic for Applications programming language.
ACL	Arbortext Command Language, a proprietary scripting language from PTC Inc.

2

Arbortext Object Model (AOM) Overview

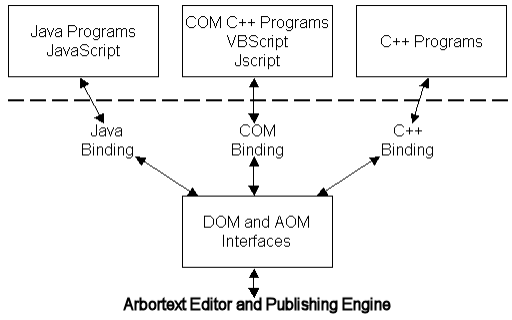
The AOM (Arbortext Object Model) delivers much of ACL's functionality available to non-ACL programmers. This includes support for the W3C DOM (Document Object Model) standard. Specifically for Arbortext Editor and Arbortext Publishing Engine, the DOM is extended with several additional interfaces, attributes, and methods.

Introduction to the Arbortext Object Model (AOM)

The AOM provides object-oriented programming access to Arbortext Editor and Arbortext Publishing Engine. The AOM supports the W3C DOM (Document Object Model) Core and Validation interfaces with extensions, and provides many additional interfaces for Arbortext-specific features that are not part of the DOM. The Arbortext extensions to the DOM use a naming convention where `A` (for Arbortext) is prepended to the DOM interface name; for example, the Arbortext extension for the DOM `Node` interface is `ANode`.

The AOM supports bindings to Java, COM (Component Object Model), and C++. The AOM also provides scripting access to its interfaces using JavaScript, JScript, VBScript, and the ACL (Arbortext Command Language).

The following diagram shows the relationship between Arbortext Editor and Arbortext Publishing Engine, the DOM and AOM interfaces, and programs or scripts accessing the DOM and AOM.



Java programs and JavaScript communicate with the DOM and AOM interfaces using the Java Binding. COM C++ programs, VBScript, and JScript communicate with the DOM and AOM interfaces using the COM Binding. C++ Programs communicate with the DOM and AOM interfaces using the C++ Binding. The DOM and AOM interfaces communicate with Arbortext Editor and the Arbortext Publishing Engine.

Introduction to the Document Object Model (DOM)

The Document Object Model (DOM) is a standards-compliant interface for examining and modifying an XML or SGML document. The DOM Level 2 specification is a recommendation of the *Worldwide Web Consortium* (W3C) comprised of several parts. Arbortext products implement the DOM Level 2 features as described in the following W3C specifications:

- Document Object Model (DOM) Level 2 Core Specification (http://www.w3.org/TR/DOM-Level-2-Core)
- Document Object Model (DOM) Level 2 Views Specification (http://www.w3.org/TR/DOM-Level-2-Views)
- Document Object Model (DOM) Level 2 Events Specification (http://www.w3.org/TR/DOM-Level-2-Events)
- Document Object Model (DOM) Level 2 Traversal and Range Specification (http://www.w3.org/TR/DOM-Level-2-Traversal-Range), range only

Arbortext also implements the W3C Recommendation Document Object Model (DOM) Level 3 Validation Specification dated 27 January 2004. (http://www.w3.org/TR/2004/REC-DOM-Level-3-Val-20040127/) The validation interfaces are implemented for both XML and SGML documents. (The DOM Level 3 Core interface **DOMConfiguration** is not implemented in this release.)

Using the DOM Support in AOM

Some considerations and limitations for using DOM through the AOM can help you determine your approach.

DOM Programming Considerations

The following programming considerations apply to all language bindings:

- Document context

  The DOM assumes that the XML document being processed is well-formed, but makes no assumptions about its validity. Because there is no way to represent validity without departing from the DOM Level 2 standard, the Arbortext Editor DOM interface ignores context checking. Therefore, it is possible for the user-written program to make a document invalid that was previously valid. However, users can context check the document once the user-written program returns control to Arbortext Editor. Alternatively, the user-written program can use the `Acl` interface to perform context checking.
- Performance issues

  The DOM allows users to create `NodeList` objects that contain pointers to every tag with a given name in a document or document subtree. Once created, a `NodeList` is dynamically updated to reflect every tag insertion or deletion. The existence of these objects is likely to slow tag insertion and deletion in Arbortext Editor. Users should delete `NodeList` objects as soon after use as practical.

DOM Limitations

The Arbortext implementation of the DOM may be used with SGML documents. Because the DOM portion of the AOM is XML- and HTML-based, features in Arbortext Editor that are available only for SGML, but not for XML, are not supported (such as `IGNORE` marked sections).

The DOM standard states that management of namespace-qualified elements and attributes will be performed without the insertion or modification of namespace-related XML attributes, at least until a document is actually written to disk. Instead, Arbortext Editor inserts `xmlns` and `xmlns:prefix` XML attributes as needed to establish and maintain namespace/prefix bindings.

Arbortext Editor does not return the document type's internal subset, if any. The `internalSubset` of the `DocumentType` interface will always return a null string.

Using the DOM with SGML Documents

The DOM is designed to support XML documents. The DOM support for SGML documents is limited to parallel support for XML. If you'll be working with SGML documents, the DOM will ignore `IGNORE` marked sections and `RCDATA` sections. If an element in an SGML document contains three sub-elements, and one of the sub-elements is an `IGNORE` marked section or an `RCDATA` section, user-written DOM programs will see only two sub-elements.

3

Custom Applications

Overview of Custom Programs and Scripts

The Arbortext Editor and Arbortext Publishing Engine installations have directory structures within them where you can place your custom scripts and programs. The `custom` and the `application` directories are described in the following sections.

The Custom Directory Structure

The *`Arbortext-path`*`\custom` directory has a subdirectory structure designed to hold your custom programs and scripts and make them automatically available during the session. At startup, these subdirectories are searched for Java, JavaScript, JScript, VBScript, ACL, and composer configuration files. You can also provide custom document types, entities, fonts, graphics, and native shared libraries and DLLs. The supported file types are automatically accessed if they reside in the appropriate subdirectory. Implementing your custom files using this approach takes advantage of the startup sequence to automatically locate your custom files. The *`Arbortext-path`*`\custom` directory and its subdirectories are explained in detail in this chapter.

The Application Directory Structure

The *`Arbortext-path`*`\application` subdirectory can contain custom applications as well as application software distributed by Arbortext. The `application` directory must have one or more uniquely named subdirectories, each containing a specific configuration file, `application.xml`, that conforms to a specific format. At startup, the `application` directory is searched for subdirectories and the presence of a valid `application.xml` file. In the uniquely named subdirectory, all subdirectories of the `custom` directory are supported. The custom application in a `application` then uses these subdirectories in the same way as the `custom` directory structure. You can also have additional subdirectories needed to support the implementation of this type of custom application. Implementing your custom application using this approach takes advantage of the startup sequence, supports delivering a completely self-contained custom application, and offers the option of setting the conditions for whether the application should be loaded. The `application` directory is also explained in this chapter.

Description of the Custom Directory Structure

When Arbortext Editor or an Arbortext PE sub-process starts, it can access custom files placed in specific directories. At startup, it automatically looks for compiled Java files (`.class` and `.jar` files), JavaScript, JScript, VBScript, ACL, document type, publishing configuration and other types of files within the *`Arbortext-path`*`\custom` directory structure.

You can have one or more `custom` directories outside the *`Arbortext-path`* install tree. To specify a path list for their locations, set the *APTCUSTOM* environment variable. The `custom` directory must be located using a file system; HTTP references are not supported.

At startup, some search paths are automatically prepended with the path to a `custom` subdirectory. Startup automatically sets some of these search paths using a symbolic variable as a path specification. You can use symbolic parameters to represent a search path in the context of the default search path, the location of the install tree, or the locale.

If a directory supports more than one type of file, the file types are processed in the following order:

- `.acl` (Arbortext Command Language) files
- `.js` (JavaScript or JScript) files
- `.class` (Java) files
- `.vbs` (VBScript) files

For each file type, its files are processed in alphabetical order by file name.

The *`Arbortext-path`*`\custom` directory is processed at startup. If you add custom applications and document types after startup, they're not recognized during the session. If you're using Arbortext Editor, it needs to be closed and restarted. If you're using Arbortext Publishing Engine, you need to stop and restart the Arbortext Publishing Engine to re-initialize the Arbortext PE sub-processes.

custom.xml File

At the top level of the `custom` directory is the `custom.xml` file. Following is the default version of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Arbortext, Inc., 1988-2009, v.4002-->
<ApplicationConfiguration
 xmlns="http://www.arbortext.com/namespace/doctypes/appcfg">
 <Information>
 <!--The following name will be shown in the New dialog
 as the category for all document types in this
 custom directory that do not specify a category.-->
```

```
 <Name>Custom Directory Name</Name>
 </Information>
</ApplicationConfiguration>
```

This file is only used when you have a custom document type in the `custom\doctypes` subdirectory, and you have not designated a category name for the document type in the associated document type configuration (`.dcf`) file's `NewDialog` element. In this case, the name in the `custom.xml` file's `Name` element is used as the **Category** name for the document type(s) in the `custom\doctypes` subdirectory in the **New Document** dialog box.

Subdirectory Structure

The following list describes each `custom` subdirectory and how it's used. Arbortext Editor and Arbortext Publishing Engine look in these directories for any references that use a relative path or have no specified path.

- `classes` subdirectory

  Holds compiled Java `.class` and `.jar` files.

  The Arbortext Editor and Arbortext Publishing Engine JVM Java class path holds a list of directories and paths to `.jar` files. Any files matching `*.jar` are prepended to the JVM Java class path. Then the `classes` parent directory is prepended, putting it first in the JVM Java class path.

  In cases where a class file occurs in more than one `.jar` file, you can extract the preferred `.class` file from its `.jar` file and place it in a subdirectory path of the `classes` directory to control which one takes precedent.

- `composer` subdirectory

  Holds publishing configuration files (`.ccf`, `.ent`, and `.xml` files) and can support a `catalog` file. Supports one level of subdirectories.

  The default path is *`Arbortext-path`*`\composer`. If there are any subdirectories of the `custom\composer` directory, those subdirectories are prepended to the publishing configuration path. Then the `custom\composer` parent directory is prepended to the path. If the `custom\composer` directory contains a `catalog` file, that directory is also prepended to the catalog path.

- `datamerge` subdirectory

  Holds data merge configuration (`.dmf`) files specifying queries and their components. The `.dmf` file structure is discussed in the *Customizer's Guide*.

- `dialogs` subdirectory

  Holds dialog files that can be accessed from custom applications, such as one that uses the AOM `Application.createDialogFromFile` method.

The *Arbortext-path*\samples\XUI\preferences\pref_exts.zip contains a sample application that adds a tab to the Preferences window as a way to extend preferences for custom applications. Refer to the readme.txt file for more information.

If there are any subdirectories of the custom\dialogs directory, those subdirectories are prepended to the dialog path. Then the custom\dialogs parent directory is prepended to the dialog path.

- ditarefs subdirectory

  Holds content referenced by DITA documents when the reference is not specified as either an absolute path name or a path name relative to the current document directory. For example, the ditarefs subdirectory could hold content referenced by topic references, content references, and so forth. Supports one level of subdirectories.

  The default DITA reference path is *Arbortext-path*\ditarefs. The DITA references path can be set in the **File Locations** category of the **Tools ► Preferences** dialog box. You can also use the set ditapath option or the *APTDITAPATH* environment variable to set the default path for DITA references. If there are any subdirectories of the custom\ditarefs directory, those subdirectories are prepended to the path. Then the custom\ditarefs parent directory is prepended to the path.

  Note

  Graphic references from DITA documents are resolved using the graphics path list.

- dictionaries subdirectory

  Holds user-defined dictionary files that can be used by the spelling checker. Supports one level of subdirectories.

  The default path is *Arbortext-path*\lib\proximity\userdict. If there are any subdirectories of the custom\dictionaries directory, those subdirectories are prepended to the dictionary path. Then the custom\dictionaries parent directory is prepended to the dictionary path.

- doctypes subdirectory

  Holds a custom catalog file and document type files. Supports one level of subdirectories. Each document type should reside in a uniquely named subdirectory of doctypes. The subdirectory should also contain a catalog file for the custom document type. A doctypes subdirectory can also contain a subset of the complete document type file set. You can place a

document type configuration file `.dcf` or stylesheets in a `\custom\doctypes\`*`doctype`* directory.

You can add a stylesheet to the list of stylesheets that displays when you make a publishing request using one of the **File ► Publish** choices. Arbortext Editor and Arbortext Publishing Engine search each `\custom\doctypes\`*`doctype`* directory and aggregate the list of stylesheets. For example, you can add stylesheets for the asdocbook built-in document type (`asdocbook`) by placing them in *`Arbortext-path`*`\custom\doctypes\asdocbook`.

If a document does not specify an Editor view stylesheet with a stylesheet association PI, Arbortext Editor will first search first the document directory, then the relevant `\custom\doctypes\`*`doctype`* directory, and finally the original location for the *`doctype`* directory.

If the subdirectory contains only a `.dcf` file, it must conform to a naming convention that expects the subdirectory and `.dcf` file name to reflect the base document type name. For example, you could customize the default asdocbook `asdocbook.dcf` file, and put it in *`Arbortext-path`*`\custom\doctypes\asdocbook\asdocbook.dcf` to override the built-in `.dcf`. Note that the document type subdirectory and file name must be the same as the default document type name for Arbortext Editor and Arbortext Publishing Engine to find all the relevant document type files.

A DCF file can reference other files, such as the `.pcf`, `demo.xml`, and `template.xml` files. Custom versions of these files can be placed with the `.dcf` in `\custom\doctypes\`*`doctype`*. If Arbortext Editor and Arbortext Publishing Engine find a `.dcf` in the `\custom\doctypes\`*`doctype`* location, relative path references are resolved by first searching the same directory as the `.dcf` and then by searching the document type directory in the original location.

The default catalog path is *`Arbortext-path`*`\doctypes`. If there are any subdirectories of the `custom\doctypes` directory that contain a `catalog` file, those subdirectories are prepended to the catalog path. Then the `custom\doctypes` parent directory is prepended to the catalog path.

You can place custom tag template files (`.tpl`) in a `custom\doctypes\`*`doctype`*`\tagtemplates` directory. The `custom\tagtemplates` directory can also be used as a more generally available location for tag templates.

Any document type from the `custom\doctypes` directory is also added to the list of available document types that are displayed in the **File ► New** dialog box.

- `entities` subdirectory

Holds file entities. Supports one level of subdirectories.

A file entity is any structurally complete document unit saved as a file. File entities commonly have an `.xml` file extension.

The default entity path is *`Arbortext-path`*`\entities`. If there are any subdirectories of the `custom\entities` directory, those subdirectories are prepended to the entity path. Then the `custom\entities` parent directory is prepended to the entities path.

- `fonts` subdirectory

  Holds custom AFM or TFM font metric files (`.afm` and `.tfm`).

  The default fonts path is *`Arbortext-path`*`\fonts`. If there are fonts in `custom\fonts`, the path is prepended. If the *APTTEXFONTS* environment variable is set, the `custom\fonts` directory is prepended to it.

- `formats` subdirectory

  Holds custom PubTex format files (`.fmt`).

  The default PubTex format path is *`Arbortext-path`*`\formats`. If there are `.fmt` files in `custom\formats`, the path is prepended. If the *APTTEXFMTS* environment variable is set, the `custom\formats` directory is prepended to it.

- `framesets` subdirectory

  Holds custom framesets for **Publish ► For Web**. Supports one level of subdirectories. Framesets are defined in the document type configuration file.

  The default frameset path is *`Arbortext-path`*`\framesets`. If there are any subdirectories of the `custom\framesets` directory, those subdirectories are prepended to the framesets path. Then the `custom\framesets` parent directory is prepended to the frameset path.

- `graphics` subdirectory

  Holds graphic files. Supports one level of subdirectories.

  The default graphics path is *`Arbortext-path`*`\graphics`. If there are any subdirectories of the `custom\graphics` directory, those subdirectories are prepended to the graphics path. Then the `custom\graphics` parent directory is prepended to the graphics path.

- `importexport` subdirectory

  Holds Arbortext Import/Export Import project files.

- `inputs` subdirectory

  Holds source files for custom macros, program fixes, or other customizations in a `custom.tmx`. Refer to Using `.tmx` files for more information.

Document type and document `.tmx` files can be placed in the `custom\doctypes` directory.

Also holds `.tex` files and source files for hyphenation exception and pattern rules in `.exc` and `.pat` files.

The default source path is *`Arbortext-path`*`\inputs`. Then the *`Arbortext-path`*`\custom\inputs` directory is prepended to it.

- `lib` subdirectory

  Holds custom versions of the `.pdfcf` PDF configuration file. The default path for `.pdfcf` files is *`Arbortext-path`*`\lib`. Then the *`Arbortext-path`*`\custom\lib` directory is prepended to it. For more information on creating `.pdfcf` files, refer to the *Customizer's Guide*.

  In addition, the `lib` subdirectory can hold `.wcf` files for custom window classes. For more information on creating `.wcf` files for window classes, refer to the *Creating custom window class preferences files* in the Arbortext Editor help.

  The `lib` subdirectory can also hold custom versions of the following files:

  `charent.cf`

  `charmap.cf`

  `installprefs.acl`

  `prted.pro`

  `pubview.cf`

  `pubview.fnt`

  `tfmfont.cf`

  `tfmscaling.cf`

  `tfontsub.cf`

  `wcharset.cf`

  `wfontsub.cf`

  `xcharset.cf`

  `xfontsub.cf`

  You can specify more than one `charent.cf` file, as the effects are cumulative. Refer to the *Setting paths for new character set files* and *APTCUSTOM environment variable* topics in the online help for more information.

The `custom\lib` directory also has `locale\`*`locale-name`* subdirectories. The default path is the appropriate locale subdirectory of *`Arbortext-path`*`\lib\locale`. The locale-specific subdirectory of the `custom\lib\locale` directory is prepended to the default locale path.

The `locale\`*`locale-name`* can hold custom versions of the `.pdfcf` PDF configuration file. For more information on creating `.pdfcf` files, refer to the *Customizer's Guide*.

Each `locale\`*`locale-name`* directory can hold custom versions of the following files:

`charent.cf`

`installprefs.acl`

`ixlang.cf`

`pubview.cf`

`pubview.fnt`

`tfmfont.cf`

`tfmscaling.cf`

`tfontsub.cf`

`wcharset.cf`

`wfontsub.cf`

`xcharset.cf`

`xfontsub.cf`

The `custom\lib` directory also has a subdirectory to hold native shared libraries for platform-specific use:

- `dll`

  Holds Windows dynamic link libraries, or DLL files (`.dll`).

  The path to this directory is prepended to the system *PATH* environment variable.

The `custom\lib` directory can have an `ixlang` subdirectory, which holds a custom `ixlang.cf` file and index mapping files like those found in *`Arbortext-path`*`\lib\ixlang`.

- `publishingrules` subdirectory

  Holds publishing rules `.prcf` files which contain definitions of publishing rules and publishing rule sets.

- `pubview` subdirectory

Holds `pubview.cf` and `pubview.fnt` files.

The default path is *`Arbortext-path`*`\pubview`. Then the *`Arbortext-path`*`\custom\pubview` directory is prepended to it.

- `scripts` subdirectory

  Holds `.acl` (Arbortext Command Language), `.vbs` (VBScript), and `.js` (JavaScript and JScript) files. Supports one level of subdirectories.

  The scripts in this directory can be called from scripts or applications in the `custom\init` directory, which is processed at startup time. Scripts placed here can be accessed using the `source` or `require` ACL commands. A customized menu item or button can call a script in `custom\scripts` when invoked.

  If there are any subdirectories of the `custom\scripts` directory, those subdirectories are prepended to the load path. Then the `custom\scripts` parent directory is prepended to the load path.

- `stylermodules` subdirectory

  Holds Arbortext Styler stylesheet modules. Any modules stored in this directory are automatically available to Arbortext Styler.

- `tagtemplates` subdirectory

  Holds `.tpl` files. You can also put custom tag templates you want associated with a particular document type into a `custom\doctypes\`*`doctype`*`\tagtemplates` directory or in the original location of the document type's *`doctype`*`\tagtemplates` directory.

  If the user clicks the **New** button from the **Tag Templates** dialog box, Arbortext Editor will use the first directory with write access for that user in the tag template path.

  If the *APTTAGTPLDIR* environment variable is set, this path is prepended to it.

- `init` subdirectory

  Holds `.acl`, `.js`, `.class`, and `.vbs` files.

  The `init` subdirectory is processed last at startup time. All files of the supported application types are executed. No nested subdirectories of `custom\init` are supported. This directory is processed after the other *`Arbortext-path`*`\custom` subdirectories so that its scripts and applications can rely on paths already established during startup.

  If you are putting custom applications on the Arbortext PE server, use the `init` directory for your custom `.acl`, `.js`, `.class` files.

In the startup process, the `custom\init` directory is processed after `_main.acl` but before `arbortext.wcf`. See the online help topic *Startup command files* for complete startup processing information.

The supported application types are:

- `.acl` (Arbortext Command Language) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

- `.js` (JavaScript or JScript) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP clients. You need to specify the JavaScript interpreter engine to use in processing `.js` files. Refer to Specifying the JavaScript Interpreter Engine on page 58 for more information.

- `.class` (Java) files

  Java `.class` files in this directory must be compiled Java classes that are not part of a named package. You can also put a `.class` file in `custom\init` that calls into a `.jar` file located in the `custom\classes` directory.

  The Java class must also implement a `public static void main(String[] args)` method, which will be called with an empty string array. If the `.class` file does not implement this method, an error is reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

- `.vbs` (VBScript) files

  Errors are reported to Arbortext Editor.

- `editinit` subdirectory

  Holds `.acl`, `.js`, `.class`, and `.vbs` files. Note that when you run Arbortext Editor with the `-c` option, any applications in this subdirectory are not executed at startup.

  All files of the supported application types are executed each time a non-ASCII document is opened for editing. Files in this directory act on a document opened in the Edit window. File in this directory act on a document opened using ACL when the `0x8000` flag is used with the `doc_open` function. File in this directory act on a document opened using AOM when the `OPEN_EDITINIT` flag is used with the **Application.openDocument** method.

  The `editinit` subdirectory is processed before any document type command files, document type instance command files, and document command files.

The supported application types are:

- `.acl` (Arbortext Command Language) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.js` (JavaScript or JScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.class` (Java) files

  Java `.class` files in this directory must be compiled Java classes that are not part of a named package. The Java class must also implement a `public static void main(String[] args)` method, which is called with an empty string array. You can put a `.class` file in `custom\init` that calls into a `.jar` file located in the `custom\classes` directory. Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

- `.vbs` (VBScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

Error Reporting for the custom\init Directory

Errors caused by mistakes in custom code in the *`Arbortext-path`*`\custom\init` directory are reported with both the error message and the name of the initialization file causing the error. Note the following:

- If Arbortext Editor is not running interactively (batch mode), no errors are reported and the errors are not logged.
- Arbortext Publishing Engine records errors and reports them to its HTTP clients in an HTML error page.
- ACL, JavaScript, and Java class errors are reported to the Arbortext Editor interface or held by Arbortext Publishing Engine to be sent to HTTP clients making requests.

Additional Information

If you are using the AOM, refer to the documentation for `Application.getCustomDirectory`. Refer to the XUI section of the *Customizer's Guide* for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following `set` command options and environment variables affect custom path search lists. They are documented in the online help.

`set catalogpath`

`set composerpath`

`set dialogspath`

`set ditapath`

`set entitypath`

`set framesetpath`

`set graphicspath`

`set javaclasspath`

`set libpath`

`set loadpath`

`set pdfconfigfile`

`set tagtemplatepath`

`set userdictpath`

Using the Custom Directory for Custom Applications

The *`Arbortext-path`*`\custom` subdirectory structure provides the means to implement custom applications. Where your application should be placed depends on the application purpose and programming language.

If you're implementing custom applications or scripts, the following information will assist you in determining the approach and location for your files:

- A custom Java program can be placed in `custom\init`, which supports a `.class` file that must implement a `public static void main (String[] args)` method. The method will be called at startup with no arguments (an empty `String` array). If an error occurs, it's reported interactively for Arbortext Editor or sent to the HTTP client for the Arbortext Publishing Engine.

  A custom Java program can also be placed in `custom\classes`, which supports `.class` or `.jar` files.

  We recommend putting Java applications in the `custom\classes` directory and calling or initializing them from the `custom\init` directory.

Paths to `.jar` files in `custom\classes` are automatically prepended to the embedded Arbortext Editor Java class path. Then the path to `custom\classes` is prepended, putting it first in the search order.

- A custom JavaScript, JScript, VBScript, or ACL application can be placed in `custom\init` or in `custom\scripts`. If you place your scripts in the `custom\scripts` directory, you can call them from a script or scripts you place in `custom\init` (which is processed at startup). Any code that exists outside a function definition in a script from `custom\init` is executed at startup time. Errors are reported if running interactively, otherwise they're suppressed.

You can create a simple JavaScript example file called `simple_init.js`. The script should contain the following line:

```
Application.alert("Hello from JavaScript");
```

Put the `simple_init.js` file in *`Arbortext-path`*`\custom\init`.

When the startup process loads scripts from `custom\init`, you will see a dialog box showing the `Hello from JavaScript` message.

Description of the Application Directory Structure

The *`Arbortext-path`*`\application` subdirectory supports installing an application into the Arbortext Editor and Arbortext Publishing Engine install trees. Arbortext Editor and the Arbortext Publishing Engine automatically search for subdirectories of the `application` directory at startup.

`Arbortext-path``\application` must contain a uniquely named subdirectory for each distributed application. Arbortext recommends using the naming pattern for a unique qualified Java class name:

`com.company-name.application-name`

Each unique subdirectory of the `application` directory must also contain an `application.xml` configuration file which describes various aspects of the application, such as its release version and supported versions of Arbortext products. At startup, Arbortext Editor and the Arbortext Publishing Engine search the `application` directory for any subdirectories containing an `application.xml` configuration file. The `application.xml` file contents provide the criteria to determine whether the application should be loaded. The `application` directory must be located using a file system; HTTP references are not supported.

Subdirectory Structure

A subdirectory of the `application` directory can be structured the same as the `custom` directory to take advantage of automatic Arbortext Editor and Arbortext Publishing Engine startup processes. For example, if the uniquely named directory contains `graphics` or `entities` directories, those directories are automatically added to the search paths constructed at startup.

An application path could be something like:
`application\`*`com.company-name.application-name`*

Refer to the Description of the custom directory structure on page 41 for the names and descriptions of each supported subdirectory.

Note

When Arbortext Editor or the Arbortext Publishing Engine constructs search paths, subdirectories of the `custom` directory take precedence over any corresponding subdirectories under the `application` directory. When search lists are constructed at startup, the first path in any search list will be the appropriate `custom` directory followed by any applicable directory under the `application` directory. For example, in constructing the graphics search path list at startup, `custom\graphics` would precede `application\com.arbortext.sample\graphics`. An `application\graphics` directory with no `application.xml` file will be ignored during startup.

When implementing a custom application using the `application` directory structure, you can add supplemental directories as needed to support your application. However, your application code must be aware of these directories and how to use them.

Application Startup File

The *`Arbortext-path`*`\doctypes\appcfg\application.xml` file provides a basic template for defining information about the custom application. You can make a copy of `doctypes\appcfg\application.xml` to use as a template to create the file that will eventually be distributed with the application. The `application.xml` file must be placed in the application's top level directory, for example:
`Arbortext-path``\application\com.`*`company.application-package-name`*
`\application.xml`

In the template `application.xml` file, you can specify a list of elements that describe the application. If the custom application determines its criteria is not met and the application is not to be loaded, then these values are ignored. The base

element for the file is the `ApplicationConfiguration` element. This element has a required attribute called *installType* that determines the type of Arbortext Editor installation for which this application is supported. The default value is `any` meaning the application is supported in both the full and compact installations of Arbortext Editor. The other supported value is `full` meaning the application is only supported in the full installation of Arbortext Editor.

The following other elements are supported in the `application.xml` file:

- `Name` (required)
- `Description`
- `LicenseNumber` is only for an application distributed by Arbortext
- `Version` (required)
- `Date`
- `Copyright`
- `Vendor`
- `RequiredApplications` is for other applications that are required for this application to run correctly. You must enter the qualified name for the application in the *qualifiedName* attribute and a human-readable name in the *name* attribute.
- `SupportedProducts`

  A `Product` element has attributes for specifying the name (required), minimum version (required), and maximum version of the Arbortext product that supports the custom application or application. The `Product` specification helps the launching Arbortext product determine whether it should load this custom application by matching criteria specified in this section.

  The name must be one or more of the following:

  - Arbortext Editor
  - Arbortext Publishing Engine
  - Arbortext Architect
  - Arbortext Editor with Styler

  The version must follow the convention used by Arbortext products, such as 5.2, 5.2 M040, or 5.3.
- `SupportedPlatforms`

  The section is reserved for future use. Windows is currently the only supported platform.
- `GlobalParameters`

`Parameter` contains `ParameterName` and `ParameterValue` elements for specifying any global variables that the application may need when it's launched.

Related Topics

If you are using ACL, refer to the following ACL function descriptions:

- `application_name function`
- `get_custom_dir function`
- `get_custom_property function`
- `get_user_property function`
- `set_user_property function`

If you are using the AOM, refer to the documentation for `Application.getCustomDirectory`. Refer to the XUI section of the *Customizer's Guide* for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following attributes from the **Application** interface are also useful:

- `haveWindows`
- `initDone`
- `isE3`
- `customProperties`
- `userProperties`
- `name`

Using the Application Directory for Custom Applications

The *`Arbortext-path`*`\application` subdirectory provides the means to implement a custom application that uses a special configuration file to determine whether it should be loaded at startup. The `application` directory uses the same principles of structure as the `custom` directory.

The *`Arbortext-path`*`\application` directory is processed at startup. If you add a custom application after startup, you must exit and restart Arbortext Editor or stop and restart the Arbortext Publishing Engine to have it recognized. You also have the option to issue the `f=init` function to re-initialize the Arbortext PE sub-processes. Refer to *Configuration Guide for Arbortext Publishing Engine* for more information.

Rules for using the `application` directory are:

- Your custom application must be contained in a uniquely named subdirectory of the `application` directory.
- You must have an `application.xml` configuration file in the uniquely named subdirectory that sets the conditions for loading the application.
- The same set of subdirectories supported by the `custom` directory are supported for the uniquely named subdirectory of the `application` directory. At startup, the supported directories are automatically detected and used in constructing search paths.
- Any other subdirectory of the `application` directory will be ignored at startup. For example, an `application\graphics` subdirectory with no `application.xml` file will be ignored during startup.

Arbortext has developed proprietary custom applications that are deployed using the `application` subdirectory structure. A uniquely named subdirectory contains all the necessary components to run an application within Arbortext Editor as well as the Arbortext Publishing Engine.

The following information will help determine an approach for a custom application.

- You can have additional subdirectories for your custom application. You are not limited to the subdirectories supported by the `custom` directory. However, these additional directories are not automatically recognized during the startup process.
- Processing each unique application's subdirectories follows the same rules for processing `custom` subdirectories. Recall that the application's subdirectories come after the `custom` subdirectories in constructing any applicable search paths for the session.
- If you decide not to use a particular supported subdirectory, you can improve performance by omitting the directory to reduce the length of a search path that would contain it.
- You can use the *APTAPPLICATION* environment variable to set the path to one or more `application` directories.
- An application should not write data to its own application directory. An application user may not have write permission access to this application directory, for example, any `C:\Program Files` directories on Windows (the location where Arbortext Editor and the Arbortext Publishing Engine are typically installed).

Deploying Zipped Customizations

You can deploy not only `custom` directories, but also `application` and content management system adapters directories in a compressed zip file. Using a zip file to distribute your customizations has the following advantages:

- You can host your customizations on a web server.

  In this case, use the HTTP or HTTPS URL to the zip file as the value for the *APTCUSTOM* environment variable.
- Your customizations will be available to users when they cannot access your network.

  If you use a shared network folder to host your customizations, users do not have access to those customizations when the network is unavailable. If you use a zip file to distribute your customizations, Arbortext Editor unzips those customizations to a directory in the Arbortext Editor cache directory (`.aptcache\zc`). At start up, Arbortext Editor checks to see whether the zip file has been updated. If it has, Arbortext Editor downloads and uncompresses the updated customizations. If not, Arbortext Editor continues to use the customizations stored in the local cache. If the network is unavailable to a user, your customizations are still available to that user in the local cache. Note that the user must also have a fixed Arbortext Editor license on their system to work away from the network.
- Network traffic might be reduced.

  Since the zip file containing your customizations is only downloaded once over the network, and then only if it has been updated, traffic on your network might be reduced. If you store your unzipped customizations in a shared network folder, Arbortext Editor might have to access that folder several times over the course of a session.
- Customizations stored in a compressed zip file are harder to change accidentally than customizations stored in a directory structure.

Note that you cannot use a zip file to distribute a customized `installprefs.acl` in the `custom\lib` directory. You can use the *APTINSTALLPREFS* environment variable to specify the location of a custom `installprefs.acl` file.

Note also that you cannot include the following font configuration files in the `lib` subdirectory of a zipped `custom` directory:

- `charent.cf`
- `wcharent.cf`
- `wfontsub.cf`
- `charmap.cf`

These files are processed before a zipped `custom` directory when Arbortext Editor starts up, so the files cannot be processed when deployed in that way.

Specifying the JavaScript Interpreter Engine

Both JavaScript and JScript files have a `.js` file extension. By default, Arbortext Editor and the Arbortext Publishing Engine interpret `.js` files as Rhino JavaScript files. You should specify the JavaScript interpreter for a JavaScript or JScript `.js` file. This is especially important if you have `.js` files of both types.

We recommend adding a comment line to your script that specifies either the Rhino JavaScript engine (the default) or the Microsoft JScript engine as shown in the following examples. The first line of your `.js` file must be a comment starting with `//`.

To specify the Rhino JavaScript interpreter:

```
// type="text/javascript"
```

To specify the Microsoft JScript interpreter:

```
// type="application/jscript"
```

The specification can be enclosed in a script tag. Both of the following examples are a valid specification for JScript:

```
// <script type="application/jscript">
// type="application/jscript"
```

You can also specify the JavaScript interpreter using the ACL `set javascriptinterpreter` command. You can specify it in an ACL file placed in the *`Arbortext-path`*`\custom\init` directory, where it will be processed at startup. For information on setting the interpreter using ACL, see the online help topic for `set javascriptinterpreter`.

II

Using the AOM

4

Using ACL with the AOM

You can access the Arbortext Object Model (AOM) from the Arbortext Command Language (ACL). Because the AOM does not currently provide all the functionality available from ACL, an AOM program may need to call ACL functions for certain types of customizations. There are several ACL functions that interface with Java, JavaScript, JScript, VBScript, and COM, which are documented in the *Arbortext Command Language Reference*. Each section in this guide that covers a specific programming or scripting language notes any language-specific binding issues.

Using the Acl Interface

The AOM provides the **Acl** interface with methods to evaluate an ACL expression (`Acl.eval`) or execute an ACL command (`Acl.execute`). Both methods take a string object as an argument. This means that any AOM object passed to ACL must be converted to a string. Likewise, an ACL type returned by `Acl.eval` is converted to a string to pass to the AOM.

The expression passed to `Acl.eval` and the command passed to `Acl.execute` are evaluated in the ACL package context of the originating ACL function that invoked the AOM method, for example, `javascript` or `js_source` for JavaScript or a `java_`*`type`* function for Java. For document type and document JavaScript and VBScript customization files automatically executed by Arbortext Editor or the Arbortext PE sub-process, this is the `main` package. If the string passed to `Acl.eval` or `Acl.execute` starts with a function call with a package prefix, then the package declaring the function is used.

Note

Be aware that the letter case to use for the **Acl** interface methods varies depending on the implementation language being used. If you are working with Java or Javascript to implement the **Acl** interface, refer to the **Acl** class Javadoc in the Arbortext Editor Help Center for the proper letter case for the **Acl** methods.

5

Using Java to Access the AOM

Java Interface Overview

Arbortext Editor and Arbortext Publishing Engine include a Java binding to the AOM. Using this binding, software developers can use the Java programming language to write applications for Arbortext Editor or the Arbortext Publishing Engine.

Arbortext Editor and the Arbortext Publishing Engine implement the Java interface using the Java Native Interface (JNI). The JNI allows Java code that runs within an embedded Java Virtual Machine (JVM) to operate with applications and libraries written in other languages such as C++. In Arbortext Editor and the Arbortext Publishing Engine, the JNI interacts specifically with the AOM.

Arbortext Editor or an Arbortext PE sub-process creates only one instance of the JVM per session and initializes it the first time a Java method is executed. The `-js` startup option may be specified when launching Arbortext Editor to cause the JVM to be initialized on startup. You can also start the JVM using the `java_init` ACL function. The JVM is unloaded when you end the current Arbortext Editor or Arbortext PE sub-process session.

There are several ACL functions of the form `java_`*`xxx`* that allow ACL programs to call a Java static method, a Java instance method, or a Java constructor, and otherwise interact with Java programs. These ACL functions are explained in Java and ACL on page 64.

Java Interface Platform Requirements

The Java interface requires access to Oracle's Java Runtime Environment (JRE), which is included in the Arbortext Editor or Arbortext Publishing Engine installation in the *`Arbortext-path`*`\bin\jre` directory.

Refer to the *Installation Guide for Arbortext Editor, Arbortext Styler, and Arbortext Architect* or *Installation Guide for Arbortext Publishing Engine* for the most recent version support information.

To use a specific JVM, you need to specify it with the `javavmpath` ACL set option. To set the maximum size of the Java Virtual Machine (JVM) memory allocation pool, use the APTJAVAVMMEMORY environment variable (sets the size of JVM on startup of Arbortext Editor) or the `javavmmemory` ACL set option.

Java and ACL

To call a Java method from ACL, use one of the following `java_`*`type`* functions.

- `java_constructor` — Calls a Java constructor.
- `java_constructor_modal` — Calls a Java constructor in a new thread.

- `java_delete` — Deletes a Java object created by `java_constructor`, `java_instance`, or `java_static`.
- `java_instance` — Calls a Java instance method.
- `java_instance_modal` — Calls a Java instance method in a new thread.
- `java_static` — Calls a Java static method.
- `java_static_modal` — Calls a Java static method in a new thread.
- `java_init` — Tests if the JVM is running and optionally initializes it.

The flow of control in the Java interface usually starts with the execution of a `java_type` ACL function.Arbortext Editor or the Arbortext PE sub-process starts its embedded Java Virtual Machine (JVM) at startup, making the distributed Java classes and user Java classes available. Java `.class` files placed in the `custom\init` directory are automatically executed without the need for the `java_type` functions.

The Java programming language supports method overloading, so several methods in a class may have the same name with different arguments. When searching for the method to invoke, Arbortext Editor or the Arbortext PE sub-process will use the first method it finds that has the correct name and correct number of arguments.

The `java_type` functions use Java reflection methods to analyze the called Java class or method before calling it, converting the arguments in the `java_type` function to the data types used by the called Java code. If you include ACL variables and function calls within your arguments, Arbortext Editor or the Arbortext PE sub-process will perform the necessary variable substitution and pass the result to the called Java code. All arguments passed are considered read-only to the called Java code; the called Java code will not change the value of any of the passed arguments.

Argument values that originate in ACL and are passed to a class or method can only be converted to a void, a Java string, or one of the supported primitive data type. The supported primitive data types are:

- int
- short
- long
- float
- double
- char
- byte

Argument values that originate as returned data from a previous call to a `java_type` function can be passed back to a Java class or method. For example, a called Java method may return a Java structure. This returned object would be

placed within the specified ACL return variable name. While this Java structure could not be used directly within ACL, you could pass it to another Java class or method by calling a `java_type` function and supplying the return variable name as an input argument.

Passing Arrays Between Java and ACL

Some ACL functions accept or return array data. Java programs that call these ACL functions will require additional coding to transfer the array data across the interface.

For example, if a Java program needs a list of the available tag names in a document, it can use the `Acl.eval` Java method to call the `tag_names` ACL function. This ACL function returns an integer for the total number of available tag names to the Java method, but it stores the array of tag names in an ACL array. To retrieve this data and make it available to the Java program, further calls to the `Acl.eval` method would be necessary. Consider the sample code that follows:

```
// This method fills a Java String array with the data
// from an ACL array
private String[] convertAclArray(String aclArrayName, \
 int aclArraySize) {
 String[] result = new String[aclArraySize];

 for (int i = 0; i < aclArraySize; i++) {
 // The first element of a Java array has index 0 but the first
 // element of an ACL array has index 1
 result[i] = Acl.eval(aclArrayName + "[" + String.valueOf(i+1)
 + "]");
}
 return result;
}
 .
 .
 .
try {
 total = Acl.eval("tag_names($arr)");
} catch (AclException e) {
 // Maybe the $arr has been defined and it is not an array
 g.drawString(e.getMessage() , 20, 60);
 return;
}
String[] names = convertAclArray("$arr", Integer.parseInt(total));
 .
 .
 .
```

Similarly, data in Java arrays need to be transferred to an ACL array before that data can be used by an ACL function.

The `java_array_from_acl` and `java_array_to_acl` ACL functions can also be used to convert certain types of arrays between ACL and Java. See the online help for details.

Java Virtual Machine (JVM) Management

By default at startup, Arbortext Editor detects and loads an installed Java Virtual Machine (JVM). You can also load the detected JVM using the `java_init` function. The JVM instance is dedicated to running Java code started from within Arbortext Editor. Arbortext Editor creates only one instance of the JVM per session. The JVM is unloaded when you end the current Arbortext Editor session.

If you choose to load another JVM, specify it with the set javavmpath ACL command, or APTJAVAVMPATH environment variable.

You can use the `set javavmmemory` ACL command to set the maximum size of the memory allocation pool before the JVM starts.

Note

If APTJAVAVMMEMORY has a value, all subsequent `set javavmmemory` commands are ignored.

By default, Arbortext Editor uses the JVM in the Java Runtime Environment (JRE) included in the Arbortext Editor installation. The JRE is located in the *`Arbortext-path`*`\bin\jre` directory. You can see the current JVM version included with Arbortext Editor by choosing **Tools ► Administrative Tools ► Java Console** to open the **Arbortext Java Console**.

Making Classes Available to the Embedded JVM

The simplest way to make your classes available to Arbortext's embedded JVM is to put them in the `custom\classes` directory. Any `.class` and `.jar` files in *`Arbortext-path`*`\custom\classes` are automatically added to the Arbortext Editor class path.

You can also use the ACL `set javaclasspath` command or the ACL `append_javaclass_path` function to set the list of directories where the embedded JVM can locate your Java classes. The default setting of `set javaclasspath` includes *`Arbortext-path`*`\custom\classes`.

The `javaclasspath` option is used only for locating non-Arbortext supplied classes. In addition to `aom.jar`, several other `.jar` files are distributed in *`Arbortext-path`*`\lib\classes` and are automatically included as part of the embedded JVM's class path.

Once the JVM has started, changes to the `javaclasspath` option or to the directories it specifies will not take effect until you exit and start a new session of Arbortext Editor or stop and restart the servlet container for the Arbortext Publishing Engine.

Making the AOM Available for Other Java Programs

If you are compiling a Java program that uses the AOM, put *`Arbortext-path`*`\lib\classes\aom.jar` in the compiler's `-classpath` argument.

Accessing the Java Console

The Java Console displays everything that a Java program writes to the Java `System.out` PrintStream and output from the JavaScript `Print()` function. The Java Console also displays the JVM version number and vendor.

> **Note**
>
> The Java Console is not a standard input (that is, `stdin`). You cannot type in the Java Console window.

For example, if you use the `java_static` function to run a Java method and that Java method executes:

```
System.out.println("Hello");
```

then `Hello` displays on the Java Console (if the Java Console is open).

If the Java Console is closed, output will be discarded.

There are two ways in which you can access the Java Console:

- Choose **Tools ► Administrative Tools ► Java Console**.
- Use the `java_console function`. You can also use this function to specify the size of the window.

AOM Packages

Arbortext Editor and the Arbortext Publishing Engine ship with Java classes for using the AOM from the Java programming language. The supplied Java classes are stored in a Java archive file *`Arbortext-path`*`\lib\classes\aom.jar` and are intended for developer use. The AOM and DOM Java classes and interfaces are stored in the following packages:

Package	Description
`com.arbortext.epic`	The core interfaces of the AOM, including the singleton **Application** and **Acl** objects.
`com.arbortext.epic.table`	The table-related interfaces for the AOM, including the **TableObject** superinterface.
`com.arbortext.epic.ui`	User interface-related interfaces for the AOM, including the **Component** superinterface.
`org.w3c.dom`	The core interfaces for the W3C Document Object Model (DOM).
`org.w3c.dom.events`	The interfaces for the W3C DOM Events specification.
`org.w3c.dom.ranges`	The interfaces for the W3C DOM Ranges specification.
`org.w3c.dom.views`	The interfaces for the W3C DOM Views specification.

All the methods in the **Application** class and the **Acl** class are class methods. Therefore you will never need an instance of the **Application** or an **Acl** object.

Note

If you inspect the `aom.jar` file, you will find additional packages (for example, **com.arbortext.epic.internal**). These additional packages are for Arbortext internal use and should not be used in your Java programs.

Your Java program should import the required AOM and DOM packages. For example, if you are writing a DOM event handler you would need to import at least the following packages:

```
import com.arbortext.epic.*;
import org.w3c.dom.*;
import org.w3c.dom.events.*;
```

See Overview on page 124 for details on using events with the AOM.

Note

The **com.arbortext.epic.ui** package defines several AOM-specific interfaces that have the same names as some in the `java.awt` package. If you import the AOM user interface package in a `.java` source file, do not also import **java.awt**.

Javadoc

Complete Java API Javadoc is delivered in the **Programming ▸ Javadoc** section of Help Center. You can also refer to the detailed documentation for each of the AOM interfaces in Interface Overview on page 189.

Compiling Your AOM Java Program

When compiling a Java program that uses the AOM, you must put *`Arbortext-path`*`\lib\classes\aom.jar` in the compiler's `-classpath` argument. For example:

```
javac -classpath "C:\Program Files\Arbortext\editor\lib\classes
\aom.jar" MyClass.java
```

The compiled program can only be run within PTC Arbortext's Java environment. Java programs running in a JVM outside of Arbortext Editor cannot use the AOM classes.

Using an IDE to create Your AOM Java Program

There are a number of Java-based Integrated Development Environments (IDE) that can be used to create AOM Java programs. The IDE must be able to find the AOM JAR file. Using Oracle's J/Developer version 3.2.2 as an example, follows these instructions:

1. Create a library

   Click on menu item **Project** followed by **Project Properties**. On the resulting dialog box, choose the **Libraries** tab and then click the **Libraries** button. On the resulting dialog box, click the **New** button and name the new library `Arbortext AOM`. In the **Class path** field on the same dialog box, specify *`Arbortext-path`*`\lib\classes\aom.jar`. Click **OK** to finish creating the library.

2. Reference the library

Return to the **Project Properties** window under the **Libraries** tab and click the **Add** button. Select `Arbortext AOM` on the resulting dialog box and click **OK** to add it to the current project.

Refer to the documentation for your IDE for instructions on a class path.

Making Classes Available to the Embedded JVM

You can use the `set javaclasspath` command or the `append_javaclass_path` function to set the list of directories where the embedded JVM can locate your Java classes. The default setting of `set javaclasspath` is empty. Regardless of whether `set javaclasspath` is set, the embedded JVM searches the distributed Java classes in *`Arbortext-path`*`\lib\classes\aom.jar`. The `aom.jar` file holds `com.arbortext.epic`, which contains the Arbortext Editor distributed Java classes that implement the AOM and DOM.

Any `.class` and `.jar` files in *`Arbortext-path`*`\custom\classes` are automatically added to the Arbortext Editor class path.

Subsequent changes to specify external Java class directories do not affect the running JVM until you exit Arbortext Editor and start a new session. Be sure to set the path to your directory before making your first Java function call.

Java Access to DOM Extensions

The AOM's extensions to DOM are represented by companion interfaces that start with the letter `A`, for example, **ANode** is the extension to the W3C **Node** interface, **ADocument** is the extension to the **Document** interface, and so on.

In Java, these interfaces can be obtained from their related objects by using the casting methods. For instance:

```
Document doc = Application.getActiveDocument();
Range r = ((ADocument)doc).getInsertionPoint();
```

Java Interface Exceptions

Several AOM and DOM methods will raise an exception if an error occurs. The following tables summarize the DOM and AOM exception classes:

DOM Exception Classes

Exception Class	Description
`DOMException`	Raised by core DOM methods.
`EventException`	Raised by DOM event methods.
`RangeException`	Raised by DOM range methods.

AOM Exception Classes

Exception Class	Description
`AclException`	Raised by methods in the **Acl** interface.
`AOMException`	Raised by general AOM methods.
`TableException`	Raised by table-related methods.
`WindowException`	Raised by **Window** and other user interface related methods.

In the Arbortext Editor Java interface, all DOM and AOM exceptions are subclasses of `java.lang.RuntimeException` and inherit the `getMessage` method from the **java.lang.Throwable** interface. The `getMessage` method can be used to retrieve an error message associated with the exception.

Most DOM and AOM exception classes define a `code` field that can be accessed to determine the numeric error code associated with the exception (the exception is the `AOMException` class). Symbolic names for the error codes listed with each exception interface description in Interface Overview on page 189 are available as class constants. For example, the following checks for a specific DOM error code (`NO_MODIFICATION_ALLOWED_ERR`):

```
try {
 node.insertBefore(newNode, refNode);
}
catch (DOMException e) {
 if (e.code == DOMException.NO_MODIFICATION_ALLOWED_ERR) {
 // document is read only
}
}
```

If your Java program does not catch an exception, its execution will be aborted and an error message will be displayed.

Accessing the Java Console

The Java Console displays everything that a Java program writes to the Java `System.out` PrintStream and output from the JavaScript `print()` function. The Java Console also displays the JVM version number and vendor.

Note

The Java Console is not a standard input (that is, `stdin`). You cannot type in the Java Console window.

For example, if a Java method executes:

```
System.out.println("Hello");
```

then `Hello` displays on the Java Console (if it is open).

If the Java Console is closed, output will be discarded.

There are two ways you can access the Java Console:

- Choose **Tools ► Java Console**.
- Use the `java_console` ACL function, which can also specify the size of the window.

Debugging Java Applications

Arbortext Publishing Engine requires you to obtain a JRE from Oracle (www.java.com) and install it. Arbortext supports the Java Platform Debugger Architecture (JPDA, see http://java.sun.com/products/jpda/),any JPDA compliant Java debugger can hook into Arbortext.

JDB can also be used to debug a Java program using two methods: the socket method and the shared memory method.

Before using JDB, ensure you have Oracle JDK version 11 or later installed on your workstation. Java debugging related DLLs and shared libraries must be accessible by the debugger. The *PATH* environment variable must include the `bin` directory of the JDK.

Compile your Java programs with the `-g` flag (for debugging).

The Socket Method

The ACL `set javadebugport` option specifies the socket port you want to use for debugging. If `javadebugport` is set to `auto`, the Arbortext Publishing Engine and Arbortext Editor will randomly select an unused socket port.

As an example, if you want to debug the `EventFlow` class, and it is located in the directory `C:\temp`, use the following steps.

1. From the Arbortext Editor command line, enter the following commands:
   ```
   set javaclasspath=C:\temp
   set javadebugport=auto
   java_console() # this loads the JVM
   eval option('javadebugport')
   ```
   Note the port number displayed in the **eval** window. For purposes of this example, assume this number was `3539`,
2. Open a shell window, navigate to the directory where your Java source resides, and enter the following command:
   ```
   jdb -connect com.sun.jdi.SocketAttach:port=3539
   ```
3. After JDB is initialized, give it a break point. For example, to break at the method `flow` of the class `EventFlow`, enter the following:
   ```
   > stop in EventFlow.flow
   ```
4. From the Arbortext Editor command line, run `EventFlow.flow` as follows:
   ```
   java_static('EventFlow','flow')
   ```
   JDB will stop at the break point and display the line of the source code where it stopped.

The Shared Memory Method

To use the shared memory method, you must set JVM arguments properly and create a name for the shared memory address.

As an example, if you want to name the shared memory address <*myaddr*>, use the following steps to debug `EventFlow.class` in `C:\temp`:

1. From the Arbortext Editor command line, enter the following commands:
   ```
   set javaclasspath=C:\temp
   set javavmargs="-Xdebug -Xrunjdwp:transport=dt_shmem,
    address=<myaddr>,server=y,suspend=n"
   # the above is one long line
   java_console()
   ```
2. Open an MSDOS shell and enter the following command:
   ```
   jdb -attach <myaddr>
   ```
3. After JDB is initialized, give it a break point. For example, to break at the method `flow` of the class `EventFlow`, enter the following:
   ```
   > stop in EventFlow.flow
   ```
4. From the Arbortext Editor command line, run `EventFlow.flow` as follows:
   ```
   java_static('EventFlow','flow')
   ```
   JDB will stop at the break point and display the line of the source code where it stopped.

Sample Java Code

Sample Java code for the Java interface is included in the *Arbortext-path*\samples\java directory. The README file in this directory provides a description of the sample code and how to invoke the sample methods. Note that you must compile the sample Java code before you can use it.

6

Using JavaScript to Access the AOM

JavaScript Interface Overview

Arbortext Editor and the Arbortext Publishing Engine include a JavaScript binding to the AOM. Using this binding, software developers can use the JavaScript programming language to write applications for Arbortext Editor and the Arbortext Publishing Engine.

Arbortext uses the Rhino open-source Java implementation from The Mozilla Organization as its JavaScript interpreter. This version of Rhino supports the JavaScript language version 1.5 and is compliant with the European Computer Manufacturers Association (ECMA) standard described in ECMA-262 Edition 3 (www.mozilla.org/js/language/E262-3.pdf).

Arbortext Editor uses the Rhino interpreter unmodified, distributed as *Arbortext-path*`\lib\classes\js.jar`. For more information about Rhino, see the *Rhino: JavaScript for Java* web page at `www.mozilla.org/rhino`. The source code for the interpreter is available at the Mozilla site at `www.mozilla.org/rhino/download.html`.

The Arbortext Object Model (AOM) interface for JavaScript is implemented on top of the Java AOM interface classes using a feature called LiveConnect. Refer to Calling Java from JavaScript on page 86 for details.

Note

The Arbortext Editor JavaScript implementation supports the DOM and Arbortext Editor AOM interfaces only. It does not support client-side JavaScript found in web browsers. In particular, there is no browser `Window` object or window global execution context. The AOM provides its own **Window** interface. By default, all JavaScript code is executed in a single global context. Arbortext Editor does not currently support other browser-specific JavaScript objects such as `Form`, `HTMLElement`, or `Location`.

JavaScript platforms

The JavaScript interface is implemented in Java, so it has the same platform requirements as the Java interface. Refer to Java Interface Platform Requirements on page 64 for more information.

JavaScript and ACL

JavaScript expressions or scripts can be called from ACL with one of the following ACL primitives:

- `javascript` — Function that evaluates a JavaScript expression and returns the result as a string.
- `js_source` — Function that reads and executes a file containing a JavaScript program.
- `js` — Command that evaluates a JavaScript expression and displays the result.
- `source` — Command that interprets files ending in `.js` as JavaScript programs to be executed when `set javascriptinterpreter` is set to `rhino`.

The flow of control in the JavaScript interface usually starts with the execution of one of these ACL functions or commands, with the exception of customization files ending in `.js`. Arbortext Editor and the Arbortext PE sub-process automatically load and execute JavaScript programs from the *`doctype`*`.js`, *`instance`*`.js`, and *`document`*`.js` files following the same rules as *`doctype`*`.acl`, *`instance`*`.acl`, and *`docname`*`.acl` files.

The JavaScript interpreter starts the first time Arbortext Editor or the Arbortext PE sub-process executes one of these ACL functions or commands or reads a `.js` customization file. Arbortext Editor and the Arbortext PE sub-process will also start the Java Virtual Machine, if necessary. You may also specify the `-jvm` and `-js` startup command options to start Java and JavaScript, respectively, when Arbortext Editor is opened.

Unlike the Java interface, only string arguments are passed from ACL to JavaScript. So any ACL argument value passed to `js_source` is converted to a string. ACL arrays must be converted to some form of delimited string (for example, as an array literal) or passed element by element to JavaScript expressions. Refer to Passing Arrays Between JavaScript and ACL on page 79 for more details.

JavaScript objects may not be returned directly to ACL. If the result of a JavaScript expression passed to `javascript` is an object, the `toString` method is invoked on the object and that value is returned by `javascript`.

Passing Arrays Between JavaScript and ACL

There are two ways to pass arrays between JavaScript and ACL, both involving the conversion of arrays to strings. The first method uses the JavaScript **Array.join** method to convert the JavaScript array to a string that is passed to the ACL `split` function.

For example, the JavaScript code

```
var jsArr = [1, 2, 3];
Acl.eval("split('" + jsArr.join() + "', aclArr, ',')");
```

converts the JavaScript array *jsArr* to the ACL array *aclArr*.

Note

ACL arrays normally start at index 1, which is the same as JavaScript index 0.

The second method uses a loop to pass the array, element by element. The **Acl.eval** call in the example above can be rewritten as:

```
for (var i = 0; i < jsArr.length; i++) {
 var ai = i + 1;
 Acl.eval("aclArr[" + ai + "] = '" + jsArr[i] + "'");
}
```

This method is slower, but isn't subject to the ACL string token limit of 4096 characters.

Similarly, there are two ways to retrieve an ACL array from JavaScript. The first method uses the ACL `join` function to concatenate the ACL array into a string that initializes a JavaScript array. For example, you can use the following ACL code to pass the ACL array created above to JavaScript:

```
javascript("var jsArr = [" . join(aclArr) . "]");
```

This method is not limited by the ACL string token limit.

You can also use a loop to retrieve the array, element by element, as shown in the following JavaScript example:

```
var count = parseInt(Acl.eval("count(aclArr)"));
var lowBound = parseInt(Acl.eval("low_bound(aclArr)"));
var jsArr = new Array(count);
for (var i = 0; i < count; i++) {
 var ai = lowBound + i;
 jsArr[i] = Acl.eval("aclArr[" + ai + "]");
}
```

This method translates the arbitrary array index bounds in an ACL array to the zero-based array index in JavaScript. It also uses the **parseInt** method to convert the Java string returned by **Acl.eval** into a JavaScript number.

Associative Arrays

The previous examples concern normal numeric indexed arrays. You can use equivalent techniques to pass associative arrays using `for/in` loops instead of the `for` loops as above. The following JavaScript example passes an associative array to ACL:

```
var jsAssoc = {one: 1, two: 2, three: 3};
for (var i in jsAssoc) {
 Acl.eval("aclAssoc['" + i + "']='" + jsAssoc[i] + "'");
}
```

You can pass an ACL associative array to JavaScript using the ACL `join` function or an ACL `for/in` loop similar to the JavaScript example. The following ACL example shows the join technique to declare a JavaScript array using object literal syntax:

```
javascript("var jsAssoc={" . join(aclAssoc,',',1) . "}")
```

Note

The ACL `join` function also works for associative arrays, and produces a result that can be used to initialize a JavaScript associative array object as in the previous example.

JavaScript Limitations

The following lists some limitations of the Arbortext Editor JavaScript implementation.

- The Mozilla Rhino JavaScript interpreter does not support the `netscape.javascript.JSObject` class as part of LiveConnect. It uses a different mechanism for accessing JavaScript objects from Java. See Requirements and Limitations at the Mozilla web page `developer.mozilla.org/en-US/docs/Web` for additional limitations of the interpreter, and the Mozilla web page `developer.mozilla.org/en-US/docs/Web/Tutorials` for a description of using JavaScript objects from Java.
- Strings returned by AOM/DOM methods are Java `String` objects and not JavaScript `String` objects. While Java `String` objects share many of the same methods as JavaScript `String` objects (for example, `charAt`, `substring`, `toLowerCase`) and can be used in string contexts, they are not equivalent. In particular, Java `String` has no `length` property; use the `length()` method instead. Also, Java `String` is not automatically converted to a number when used in a numeric context. To explicitly convert a Java `String` to a number when appropriate, use the `parseInt` or `parseFloat` function.

  To perform JavaScript-style string manipulations on a Java `String` returned by the AOM, convert the string to a JavaScript `String` by concatenating it with a null string. For example:

  ```
  var jsStr = doc.documentElement.tagName + "";
  ```

JavaScript Language Extensions

The Arbortext Editor JavaScript implementation includes a few non-standard extensions, modeled on similar features provided by the Rhino Shell. The Rhino Shell is a standalone utility from Mozilla that runs JavaScript programs.

Function	Description
`defineClass(`*`javaclass`*`)`	This global function defines a JavaScript class from the Java class specified by *`javaclass`*. The Java class file must be in the class path set for the Java Virtual Machine embedded in Arbortext Editor, for example, by including the `.class` file in the *`Arbortext-path`*`\custom\classes` directory. *`javaclass`* must implement the **org.mozilla.javascript.Scriptable** interface or extend the **org.mozilla.javascript.ScriptableObject** class. See the *Rhino documentation* at the Mozilla web page (`www.mozilla.org/rhino/doc.html`) for details.
`implementationVersion()`	This global function returns the JavaScript interpreter implementation version as a string encoding the product name, language version, release number, and date.
`importClass(`*`javaclass`*`)`	This global function will “import” the Java class *javaclass* by making its unqualified name available as a property of the top-level scope.
`importPackage(`*`javapackage`*`)`	This global function will “import” all the classes of the Java package *javapackage* by searching for unqualified names as classes qualified by the given package. This is similar to the Java import statement.

Function	Description
	Note If this function is evaluated in the global scope, then the unqualified names are available to all JavaScript code subsequently executed in the shared scope.
`load(`*`filename, ...`*`)`	This global function will load and execute the JavaScript source file given by the *filename* argument. Multiple file name arguments may be specified and *filename* can be a URL. If *filename* is not an absolute path or URL, the list of directories is the list in *loadpath* parameter of the `setOption` method, described in AOM set Options Overview on page 741. If *filename* is not found relative to the current directory and is not an absolute path, the list of directories specified in the Arbortext Editor (or the Arbortext Publishing Engine) *loadpath* parameter is searched to locate the JavaScript source file.

Function	Description
`print(`*`expr`*`)`	This global function evaluates the expression *expr* and prints the string value of the result to the Java Console. If the Java Console is not open, the output is discarded. The `print` function supplies a trailing new line character, so each call to `print()` ends a line.
`quit()`	This global function terminates the current script execution. It is provided so sample Rhino JavaScript scripts can be run unmodified within Arbortext Editor and the Arbortext Publishing Engine. This function is implemented by throwing a special **JavaScriptException** object; if `quit()` is used inside a try block with a catch, it will not function as expected.

JavaScript Global Objects

The Arbortext JavaScript implementation provides several global objects available to all JavaScript scripts. The **Application** and **Acl** objects are instances of the AOM **Application** and **Acl** interfaces. Only one object for each interface exists in a Arbortext Editor or Arbortext PE sub-process session.

Object	Description
Application	This global object implements the **Application** interface that provides access to all other DOM and AOM objects except for the **Acl** interface.
Acl	This global object implements the **Acl** interface that provides access to ACL (Arbortext Command Language).
AclException	This is an instance of the class `AclException`, raised by some **Acl** interface methods.
DOMException	This is an instance of the class `DOMException`, raised by some DOM interface methods.

Object	Description
EventException	This is an instance of the class `EventException`, raised by some DOM **Event** interface methods.
RangeException	This is an instance of the class `RangeException`, raised by some DOM **Range** interface methods.
TableException	This is an instance of the class `TableException`, raised by some **Table** interface methods.
WindowException	This is an instance of the class `WindowException`, raised by some UI interface methods.
`arguments`	This global array contains the arguments passed to the `js_source` ACL function as the *args* parameter. The array will have zero length if no arguments were passed, or if the JavaScript code was executed by the `javascript` ACL function.
`environment`	This global object provides access to Java System properties. Accessing an environment property `name` results in a call to `java.lang.System.getProperty("name")`. Setting a property `name` to `value` results in a call to `java.lang.System.getProperties().put("name", "value")`. For example: `environment["user.dir"] = "c:\\temp"` changes the java user directory system property.

Calling Java from JavaScript

The Mozilla Rhino JavaScript interpreter bundled with Arbortext Editor provides a mechanism called LiveConnect that lets you use Java classes and methods from JavaScript. The Arbortext Object Model (AOM) classes are written in Java and made available in JavaScript by LiveConnect.

LiveConnect manages the Java to JavaScript communication, including conversion of data types. JavaScript: The Definitive Guide, written by David Flanagan and published by O'Reilly, discusses this subject. There are some limitations with LiveConnect and the AOM, as noted in JavaScript Limitations on page 81.

Rhino also supports defining new JavaScript classes by writing Java code that extends the **org.mozilla.javascript.ScriptableObject** class. The JavaScript function `defineClass` makes such classes available to JavaScript. Refer to the Rhino documentation at the Mozilla web page (`www.mozilla.org/rhino/doc.html`) for details.

With LiveConnect, Java packages are represented in JavaScript by the `JavaPackage` class. You can access the Java classes provided with the JVM embedded in Arbortext Editor, plus those found in the Java class path (as specified by the *javaclasspath* parameter of the `setOption` method, described in AOM set Options Overview on page 741) from the top-level JavaPackage object `Packages`. This includes the standard Java system classes (for example, `Packages.java.lang.System`) and the packages provided by Arbortext (for example, `Packages.com.arbortext.epic`, `Packages.org.w3c.dom`), and the JavaScript interpreter (`Packages.org.mozilla.javascript`). As a convenience, the classes in the `java` package can be referred to directly without the `Packages` qualifier, for example, `java.lang.System` and `java.lang.awt.Frame`.

Note

The Java Swing classes are in the `javax` package, so you must fully qualify the package name (`Packages.javax.swing`) to use Swing classes.

The global object `Application` is a shortcut for the `Packages.com.arbortext.epic.Application` JavaClass. Similarly, the global object `Acl` is a shortcut for the `Packages.com.arbortext.epic.Acl` JavaClass.

The following JavaScript example uses the standard Java AWT classes to create and display a dialog box.

Note

Since no event handling is specified in this example, the dialog box cannot be dismissed.

```
function hello()
{
 var f = new java.awt.Frame("Hello World");
 var ta = new java.awt.TextArea("hello, world", 100, 200);
 f.add("Center", ta);
 f.pack();
 f.show();
}
hello();
```

A more complicated example with event handling is included with the Arbortext distribution. Refer to Sample JavaScript Code on page 88 for details.

JavaScript Interface Error Handling

Errors When Executing JavaScript

When executing JavaScript programs, Arbortext Editor displays error messages if there are problems when starting the JavaScript interpreter, in the embedded Java Virtual Machine (JVM), or if the JavaScript interpreter reports an exception. If the JavaScript interpreter reports an exception, Arbortext Editor displays a message such as “The Java method *name* has thrown an exception.” If you use the ACL function `javascript` to invoke the JavaScript interpreter, *name* is `eval`; if you use the ACL function `js_source`, *name* is `source`.

The JavaScript exception message is sent to the Java Console if it is open; otherwise, it is discarded. When developing JavaScript applications, choose **Tools ► Java Console** to open the Java Console and view exception messages.

For JavaScript code executed by reading a `.js` file, the JavaScript exception report includes a traceback showing the file name and line number of each function active at the time of the error. The traceback also lists Java methods for the JavaScript interpreter, which can be ignored.

Exception Handling

JavaScript provides exception handling with try/catch statements. Since JavaScript is implemented using the Java interface, it supports all the DOM and AOM exception classes summarized in Java Interface Exceptions on page 71 and defined in Interface Overview on page 189. Most exception classes define a

numeric error code attribute named `code` and message attribute named `message`. The symbolic names for the error codes listed with each exception interface description are available for the global exception objects listed in JavaScript Global Objects on page 84. For example,

```
try {
 node.insertBefore(newNode, refNode);
}
catch (e) {
 if (e.code == DOMException.NO_MODIFICATION_ALLOWED_ERR) {
 Application.alert("Document is read only");
}
 else {
 Application.alert("Error: " + e.code +
 " Message: " + e.message);
}
}
```

Specifying the Interpreter for .js Files

Arbortext Editor supports two JavaScript interpreters. You should specify which interpreter to use to process your `.js` files. You can include a special comment as the first line of the file. If the first line of the `.js` file using either form specified in the following examples, then the Rhino JavaScript interpreter will be used.

```
// type="text/javascript"
```

or

```
// <script type="text/javascript">
```

You can also specify the interpreter with the ACL `set javascriptinterpreter` command. However, we recommend using the commenting technique as it ensures proper handling of your `.js` files regardless of the *javascriptinterpreter* setting.

Sample JavaScript Code

Sample JavaScript code that uses the JavaScript AOM interface is included in the *Arbortext-path*`\samples\javascript` directory. The `readme.txt` file in this directory provides a description of the sample code and how to invoke the sample scripts. The samples include examples of using the DOM to manipulate the active document, registering DOM Event handlers, using Java AWT classes, and transferring arrays between JavaScript and ACL.

There is a sample from the Mozilla Rhino distribution that implements a JavaScript `File` class in Java and an example script, `jsdoc.js`, that uses the `defineClass` JavaScript extension to define the `File` class.

Refer to Rhino Examples at the Mozilla web page (`www.mozilla.org/rhino/examples.html`) for additional sample JavaScript scripts.

7

Using COM to Access the AOM

COM Interface Overview

Arbortext Editor includes a Component Object Model (COM) binding to the AOM. Using this binding, developers on Windows platforms can write programs that use COM to access the AOM or DOM functions supported in Arbortext Editor.

COM should be installed on all Windows systems that are running Arbortext Editor. It is unlikely that your Windows systems will not have COM already installed on them.

When acting as a COM server, Arbortext Editor registers an `Epic.Application` COM class which implements the **_Application*N*** interface (for example, **_Application6** — consult the type library for the correct interface version), an `Epic.Acl` COM class which implements the **IAcl3** interface, a number of `DOM`*`xxx`* classes which implement their respective **IDOM*xxx*** interfaces, and many other *`xxx`* classes that implement their respective **I*xxx*** AOM interfaces.

If you are trying to use COM among different machines, you will need to install DCOM (Distributed Component Object Model). Extensive information on both COM and DCOM is available from the Microsoft Developers Network (MSDN) web site at msdn.microsoft.com.

The Arbortext Editor COM interface to the DOM portion of AOM uses the COM binding defined by Microsoft with changes for DOM Level 2 and Arbortext extensions. However, Microsoft has made several significant extensions to the DOM that are not supported by Arbortext. The definition of the COM classes and methods that Arbortext Editor exports is contained in the type library that is part of the *`Arbortext-path`*`\bin\editor.exe` binary. Developers can use a variety of tools to inspect this type library.

The type library defines multiple versions of many interfaces. When an interface is extended for a given Arbortext Editor or Arbortext Publishing Engine release, a new version of the interface is defined with the version number incremented. For example, the **_Application3** interface was introduced with Epic Editor and E3 4.3.

Arbortext Editor or an Arbortext PE sub-process does not need to be running for it to be available to COM. If Arbortext Editor or an Arbortext PE sub-process is not running when a call is made to the Arbortext Editor COM server, it will automatically load and run in the background while servicing the COM call. If a user then uses the Windows user interface to start a Arbortext Editor session, the invisible instance that was running exclusively as a COM server automatically becomes visible and available to the user.

Registering and Unregistering Arbortext Editor as a COM Server

When you install Arbortext Editor, the `setup` program automatically registers PTC Arbortext Editor as a COM server. The uninstall program will unregister Arbortext Editor as a COM server.

Starting with release 5.4, Arbortext Editor also automatically checks at startup to see whether the application is registered as a COM server. If Arbortext Editor finds that it is not registered as a COM server, it performs a COM registration for Arbortext Editor itself and all of its installed components as part of the startup process. This check can be disabled with the *APTNOCOMCHECK* environment variable. If the automatic registration fails for some reason (usually because the user does not have administrator privileges), Arbortext Editor still opens but displays an error message first saying that this version is no longer configured correctly. In this case, some Arbortext Editor components might not be available. You can keep Arbortext Editor from opening in this case with the *APTFAILIFNOCOM* environment variable.

If you run a version of Arbortext Editor earlier than 5.4 on the same system with your current version, you might encounter problems with the earlier version's COM registration due to the new automatic COM registration. You can obtain a utility called `register.bat` from PTC Technical Support that will correctly register releases of Arbortext Editor prior to 5.4. For more information, search the Technical Support knowledge base for TPI 144503.

You can manually register or unregister a PTC Arbortext Editor installation at any time by running Arbortext Editor with the `-RegServer`, `-UnregServer`, or `-UnregAnyServer` startup command options. In the examples that follow, the first path to the `editor.exe` binary is for 64-bit installations, and the second path is for 32-bit installations.

```
Arbortext-path\bin\x86\editor.exe -RegServerArbortext-path\bin
\x86\editor.exe -RegServer
Arbortext-path\bin\x64\editor.exe -RegServerArbortext-path\bin
\x86\editor.exe -RegServer
```

Registers a specific Arbortext Editor installation as a COM server.

```
Arbortext-path\bin\x86\editor.exe -UnregServerArbortext-path\bin
\x86\editor.exe -UnregServer
Arbortext-path\bin\x64\editor.exe -UnregServerArbortext-path\bn
\x86\editor.exe -UnregServer
```

Unregisters a specific Arbortext Editor installation as a COM server. Note that the `-UnregServer` option will not remove the `editor.exe` COM server entry in the registry, unless the Arbortext Editor installation you are running matches the Arbortext Editor installation listed as the current `editor.exe` COM server.

```
Arbortext-path\bin\x86\editor.exe -UnregAnyServerArbortext-path\bin
\x86\editor.exe -UnregAnyServer
Arbortext-path\bin\x64\editor.exe -UnregAnyServerArbortext-path\bin
\x86\editor.exe -UnregAnyServer
```

Unregisters any version of Arbortext Editor on the system as a COM server, not just the installation for which you are using the option.

Accessing COM Using JScript or VBScript

You can access the AOM in JScript and VBScript using the COM interface. The Arbortext Editor **Application** and **Acl** objects are exposed to the script automatically as global objects when using the built-in script interpreters.

You can access external third-party COM objects using the JScript `ActiveXObject` object or the VBScript `CreateObject` and `GetObject` functions. Microsoft Excel is an example of a COM server which can be accessed from Arbortext Editor. For example, to launch Microsoft Excel using JScript, use the following statement:

```
var xl = new ActiveXObject("Excel.Application");
```

To launch it using VBScript, use:

```
Dim xl
set xl = CreateObject("Excel.Application")
```

Both examples provide access to Excel's **Application** object, which is different from the Arbortext Editor **Application** object. (If you were running a script outside the built-in interpreter, for example, using Excel VBA, you would need to create an instance of the Arbortext Editor **Application** object using `Epic.Application`.)

Extensive documentation on JScript and VBScript is available from the Microsoft Developers Network (MSDN) web site at msdn.microsoft.com. Search for the topic “Windows Script”. Documentation on how to use a COM server, such as Excel, is provided by the software vendor. In the case of Microsoft Office products, the VBA (Visual Basic for Applications) documentation is the primary source of information on the COM objects exposed in each Microsoft Office application.

COM Objects and ACL

You can use ACL (Arbortext Command Language) to call most COM (Component Object Model) objects which export the `IDispatch` interface and which include a type library.

You can use this functionality, for example, to invoke an application or DLL written in Visual Basic. Such an external application can, in turn, invoke Arbortext Editor or an Arbortext PE sub-process using its COM interface to access or change a document. Keep in mind that calling COM objects from VBScript or JScript scripts is more straightforward than calling COM objects from ACL (refer to Accessing COM Using JScript or VBScript on page 92).

ACL includes a set of functions to support COM calls: `com_attach`, `com_call`, `com_prop_get`, `com_prop_put`, and `com_release`.

Use the `com_attach` function to attach to a COM object and return a handle that can be used to invoke the object. After a successful `com_attach`, you can use the object handle to make calls to `com_call`, `com_prop_get`, or `com_prop_set` to invoke a method or get or set a property in a COM interface. Use the `com_release` function to release an object attached by `com_attach` or one returned by another interface. These functions are documented in the *Arbortext Command Language Reference*.

Arbortext Editor and the Arbortext PE sub-process use the type library associated with a COM interface to determine the type of each argument and the return value of a method or property invoked using an ACL function. This makes it possible, for example, to pass ACL variables to COM methods that expect parameters passed by reference and have the COM object return results to ACL by changing the value of the variable.

Arbortext Editor and Arbortext Publishing Engine have some restrictions and limitations in their support for calling COM interfaces, many of which are inherent to ACL:

- Named arguments are not supported.
- Arguments can be omitted only at the end of the argument list
- You cannot pass an ACL array to a COM interface as an array. You can pass a member of an ACL array as an individual argument.
- A called COM interface function can't return an array and have it converted into an ACL array.
- You cannot use the other information in a type library (such as `enum` definitions) in ACL.
- There is no implicit support for the implied `Value`, `_NewEnum`, or `Evalute` methods and properties even though it may be possible to call them explicitly.

COM Error Handling

All of the Arbortext Editor COM interfaces support the **ErrorInfo** COM interface and use it to pass error messages to the client if the called method fails. All supplied methods return an `HRESULT` which indicates success or failure and the general nature of the failure. Developers can use standard COM practices to retrieve error codes and error messages.

The DOM specification indicates that several methods will raise an exception upon certain types of failure. This is also the case for several AOM methods. Since the COM interface doesn't support exceptions, these failures will be turned into `HRESULT` return values. The specific value returned for a given exception

can be found in the type library for the *Arbortext-path*\bin\editor.exe binary. They're also presented in the tables that follow. The general rule is that these exceptions will be returned as DOM_E_*YYY*_ERR for the **DOMException**, **EventException** and **RangeException** errors, TABLE_E_*YYY*_ERR for **TableException** errors, WINDOW_E_*YYY*_ERR for **WindowException** errors, and EXECUTE_E_*YYY* for **AclException** errors.

The following tables list the COM error codes and values for each range of errors. See the exception interface definitions in Interface Overview on page 189 for the exception codes and their meanings.

DOM Error Codes

Error Code	Value
DOM_E_INDEX_SIZE_ERR	0x80042101
DOM_E_DOMSTRING_SIZE_ERR	0x80042102
DOM_E_HIERARCHY_REQUEST_ERR	0x80042103
DOM_E_WRONG_DOCUMENT_ERR	0x80042104
DOM_E_INVALID_CHARACTER_ERR	0x80042105
DOM_E_NO_DATA_ALLOWED_ERR	0x80042106
DOM_E_NO_MODIFICATION_ALLOWED_ERR	0x80042107
DOM_E_NOT_FOUND_ERR	0x80042108
DOM_E_NOT_SUPPORTED_ERR	0x80042109
DOM_E_INUSE_ATTRIBUTE_ERR	0x8004210A
DOM_E_INVALID_STATE_ERR	0x8004210B
DOM_E_SYNTAX_ERR	0x8004210C
DOM_E_INVALID_MODIFICATION_ERR	0x8004210D
DOM_E_NAMESPACE_ERR	0x8004210E
DOM_E_INVALID_ACCESS_ERR	0x8004210F
DOM_E_VALIDATION_ERR	0x80042110
DOM_E_UNSPECIFIED_EVENT_TYPE_ERR	0x80042148
DOM_E_BAD_BOUNDARYPOINTS_ERR	0x80042141
DOM_E_INVALID_NODE_TYPE_ERR	0x80042142
DOM_E_NO_SCHEMA_AVAILABLE_ERR	0x80042647

Table Interface Error Codes

Error Code	Value
TABLE_E_TABLE_OPERATION_FAILED_ERR	0x80042301
TABLE_E_TABLE_INVALID_INDEX_ERR	0x80042302
TABLE_E_TABLE_INVALID_DIRECTION_ERR	0x80042303

Table Interface Error Codes (continued)

Error Code	Value
TABLE_E_TABLE_INVALID_ORIENTATION_ERR	0x80042304
TABLE_E_TABLE_INVALID_SPAN_ERR	0x80042305
TABLE_E_TABLE_INVALID_PARAMETER_ERR	0x80042306
TABLE_E_TABLE_INVALID_ATTRIBUTE_ERR	0x80042307

Window Interface Error Codes

Error Code	Value
WINDOW_E_WINDOW_NOT_SUPPORTED_ERR	0x80042401
WINDOW_E_WINDOW_HIERARCHY_REQUEST_ERR	0x80042402
WINDOW_E_WINDOW_WRONG_WINDOW_ERR	0x80042403
WINDOW_E_WINDOW_NOT_FOUND_ERR	0x80042404
WINDOW_E_WINDOW_INVALID_COLOR_ERR	0x80042405
WINDOW_E_WINDOW_INVALID_MODIFICATION_ERR	0x80042406
WINDOW_E_WINDOW_NO_MODIFICATION_ ALLOWED_ERR	0x80042407

Acl.Execute Error Codes

Error Code	Value
EXECUTE_E_PARSE_FAILURE	0x80042200
EXECUTE_E_ERROR	0x80042201
EXECUTE_E_INTERNAL_ERROR	0x80042202

JScript maps the COM errors to the `Error` object, and VBScript maps the COM errors to the `Err` object. See JScript Exception Handling on page 102 and VBScript Error Handling on page 108 for details.

Sample COM Code

Sample Visual Basic and Visual C++ code that uses the COM interface is included in the *`Arbortext-path`*`\samples\com` directory. The `Readme` file in this directory provides details on the samples.

8

Using JScript to Access the AOM

JScript Interface Overview

Arbortext Editor and the Arbortext Publishing Engine include a JScript binding to the AOM. Using this binding, software developers can use the JScript programming language to write applications for Arbortext Editor and the Arbortext Publishing Engine.

Arbortext uses Microsoft Windows Script (or ActiveScript) as the JScript interpreter. This script engine is represented primarily by the system files `jscript.dll` and `scrrun.dll` which are typically installed by Microsoft Windows, Internet Explorer, and the Windows Script Host upgrades available from the Microsoft Developers Network (MSDN). Arbortext recommends Windows Script Version 5.6, which is free from the Microsoft web site at: msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/letintro.asp.

Note

JScript versions prior to 5.0 shipped with Windows 98 have not been tested.

The AOM interface and the DOM interface for JScript are implemented using the PTC Arbortext COM interface. Access to external COM servers is implemented through standard COM interfaces used by the Microsoft script engines.

Note

By default, all JScript code is executed in a single global context, in a namespace called `EpicJS`. A JScript instance can create nested JScript instances which use unique namespaces. See the description of the `createScriptContext` method for the AOM Application object in createScriptContext method on page 285.

JScript Platforms

The JScript interface is a Windows-only technology, available on Microsoft Windows 2000 and Windows XP.

JScript with ACL

JScript expressions or scripts can be called from ACL with one of the following ACL primitives:

- `jscript` — Function that evaluates a JScript expression and returns the result as a string.
- `js` — Command that evaluates a JScript expression and displays the result.
- `source` — Command that interprets files ending in `.js` as JavaScript programs to be executed when `set javascriptinterpreter` is set to `jscript`.

The flow of control in the JScript interface usually starts with the execution of one of these ACL functions or commands, with the exception of customization files ending in `.js`. Arbortext Editor and the Arbortext PE sub-process automatically load and execute JScript programs from the *`doctype`*`.js`, *`instance`*`.js`, and *`document`*`.js` files following the same rules as *`doctype`*`.acl`, *`instance`*`.acl`, and *`docname`*`.acl` files.

The JScript interpreter starts the first time Arbortext Editor or the Arbortext PE sub-process executes one of these ACL functions or commands or reads a `.js` customization file. Arbortext Editor and the Arbortext PE sub-process will also start the Java Virtual Machine, if necessary. You may also specify the `-jvm` and `-js` startup command options to start JScript when Arbortext Editor is opened.

Unlike the Java interface, only string arguments are passed from ACL to JScript. ACL arrays must be converted to some form of delimited string (for example, as an array literal) or passed element by element to JScript expressions. Refer to Passing Arrays Between JavaScript and ACL on page 99 for more details.

JScript objects may not be returned directly to ACL. If the result of a JScript expression passed to `javascript` is an object, the `toString` method is invoked on the object and that value is returned by `javascript`.

Passing Arrays Between JavaScript and ACL

There are two ways to pass arrays between JScript and ACL, both involving the conversion of arrays to strings. The first method uses the JScript **Array.join** method to convert the JScript array to a string that is passed to the ACL `split` function.

For example, the JScript code

```
var jsArr = [1, 2, 3];
Acl.eval("split('" + jsArr.join() + "', aclArr, ',')");
```

converts the JScript array *jsArr* to the ACL array *aclArr*.

Note

ACL arrays normally start at index `1`, which is the same as JavaScript index `0`.

The second method uses a loop to pass the array, element by element. The **Acl.eval** call in the previous example can be rewritten as:

```
for (var i = 0; i < jsArr.length; i++) {
 var ai = i + 1;
 Acl.eval("aclArr[" + ai + "] = '" + jsArr[i] + "'");
}
```

This method is slower, but isn't subject to the ACL string token limit of 4096 characters.

Similarly, there are two ways to retrieve an ACL array from JScript. The first method uses the ACL `join` function to concatenate the ACL array into a string that initializes a JScript array. For example, you can use the following ACL code to pass the ACL array created above to JScript:

```
javascript("var jsArr = [" . join(aclArr) . "]");
```

This method is not limited by the ACL string token limit.

You can also use a loop to retrieve the array, element by element, as shown in the following JScript example:

```
var count = parseInt(Acl.eval("count(aclArr)"));
var lowBound = parseInt(Acl.eval("low_bound(aclArr)"));
var jsArr = new Array(count);
for (var i = 0; i < count; i++) {
 var ai = lowBound + i;
 jsArr[i] = Acl.eval("aclArr[" + ai + "]");
}
```

This method translates the arbitrary array index bounds in an ACL array to the zero-based array index in JScript. It also uses the **parseInt** method to convert the Java string returned by **Acl.eval** into a JScript number.

Associative arrays

The previous examples concern normal numeric indexed arrays. You can use equivalent techniques to pass associative arrays using `for/in` loops instead of the `for` loops as above. The following JScript example passes an associative array to ACL:

```
var jsAssoc = {one: 1, two: 2, three: 3};
for (var i in jsAssoc) {
 Acl.eval("aclAssoc['" + i + "']='" + jsAssoc[i] + "'");
}
```

You can pass an ACL associative array to JScript using the ACL `join` function or an ACL `for/in` loop similar to the JScript example. The following ACL example shows the join technique to declare a JScript array using object literal syntax:

```
javascript("var jsAssoc={" . join(aclAssoc,',',1) . "}")
```

Note

The ACL `join` function also works for associative arrays, and produces a result that can be used to initialize a JavaScript associative array object as in the previous example.

JScript Limitations

Some limitations of the Arbortext JScript implementation are:

- JScript is not case-sensitive. Rhino JavaScript is case-sensitive. AOM and DOM compatiblity between JScript and JavaScript files requires the script author to comply with the capitalization of methods and attributes described in this guide.
- The AOM and DOM constants are not defined in the global context. They must be defined inline in JScript files to be referenced by variable name.

AOM Interfaces Specific to JScript

By default, JScript instances run in a single global context, or namespace, called `EpicJS`. The AOM includes JScript-specific features related to the **ScriptContext** interface:

- `createScriptContext`—allows scripts to create and run nested scripts in the global namespace (`EpicJS`) or in a user-defined context or namespace.
- `getScriptContext`—retrieves a reference to any running script context by namespace.

See the descriptions in Application interface on page 273 and ScriptContext interface on page 629 for more information.

JScript Global Objects

The Arbortext JScript implementation provides several global objects available to all JScript scripts. The **Application** and **Acl** objects are instances of the AOM **Application** and **Acl** interfaces. Only one object for each interface exists in a Arbortext Editor session.

Object	Description
Application	This global object implements the **Application** interface that provides access to all other DOM and AOM objects except for the **Acl** interface.
Acl	This global object implements the **Acl** interface that provides access to ACL (Arbortext Command Language).

JScript Exception Handling

JScript provides exception handling with try/catch statements. JScript is implemented using the COM interface, so it does not support the DOM and AOM exception classes. All exceptions are mapped to the JScript `Error` global object. The COM error code values listed in COM Error Handling on page 93 are available using the `number` property of the `Error` object. The message associated with the exception is available using the `description` property. For example:

```
try {
 doc.insertBefore(doc, doc); // this is invalid
}
catch(e) {
 Application.alert("Error: " + (e.number&0xffff) +
 " Description: " + e.description);
}
```

Specifying the Interpreter for `.js` Files

Arbortext Editor supports two JavaScript interpreters on Windows. You should specify which interpreter to use to process your `.js` files. You can include a special comment as the first line of the file. If the first line of the `.js` file contains a comment using either form specified in the following examples, then the Microsoft JScript interpreter will be used.

```
// application="text/jscript"
```

or

```
// <script application="text/jscript">
```

You can also specify the interpreter with the ACL `set javascriptinterpreter` command. However, we recommend using the commenting technique as it ensures proper handling of your `.js` files regardless of the *javascriptinterpreter* setting.

Sample JScript Code

Sample JScript code that uses the JScript AOM interface is included in the *Arbortext-path*\samples\jscript directory. The readme.txt file in this directory provides a description of the sample code and instructions for invoking the sample scripts. Examples show how to use the DOM to manipulate the active document, register DOM Event handlers, and transfer arrays between JScript and ACL. The JScript examples are ported from the corresponding Rhino JavaScript samples of the same name.

9

Using VBScript to Access the AOM

VBScript Interface Overview

Arbortext Editor and the Arbortext Publishing Engine include a VBScript binding to the AOM. Using this binding, software developers can use the VBScript programming language to write applications for Arbortext Editor and the Arbortext Publishing Engine.

Arbortext uses Microsoft Windows Script (or ActiveScript) as the VBScript interpreter. This script engine is represented primarily by the system files `vbscript.dll` and `scrrun.dll` which are typically installed by Microsoft Windows, Internet Explorer, and the Windows Script Host upgrades available on the Microsoft Developers Network (MSDN). Arbortext recommends the most recent version of Windows Script, Version 5.6, which is free from the Microsoft web site at: msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/letintro.asp.

Note

VBScript versions prior to 3.1 shipped with Windows 98 have not been tested.

The AOM interface and the DOM interface for VBScript is implemented via Arbortext's COM interface. Access to external COM servers is implemented through standard COM interfaces used by the Microsoft script engines.

Note

By default, all VBScript code is executed in a single global context, in a namespace called `EpicVBS`. A VBScript instance can create nested VBScript instances which use unique namespaces. See the `createScriptContext` method for the AOM Application object in createScriptContext method on page 285.

VBScript Platforms

The VBScript interface is a Windows-only technology, available on Windows 2000 and Windows XP.

VBScript and ACL

VBScript expressions or scripts can be called from ACL with one of the following ACL primitives:

- `vbscript` — Function that evaluates a VBScript expression and returns the result as a string.
- `source` — Command that interprets files ending in `.vbs` as JScript programs to be executed.

VBScript Limitations

Some limitations of the Arbortext VBScript implementation are:

- VBScript is not case-sensitive.
- The AOM and DOM constants are not defined in the global context. They must be defined inline in VBScript files to be referenced by variable name.

AOM Interfaces Specific to VBScript

By default, VBScript instances run in a single global context, or namespace, called `EpicVBS`. The AOM includes VBScript-specific features related to the **ScriptContext** object:

- `createScriptContext` — allows scripts to create and run nested scripts in the global namespace (`EpicVBS`), or in a user-defined context or namespace.
- `getScriptContext` — retrieves a reference to any running script context by namespace.

See the descriptions in Application interface on page 273 and ScriptContext interface on page 629 for more information.

VBScript Global Objects

The Arbortext VBScript implementation provides several global objects available to all VBScript scripts. The **Application** and **Acl** objects are instances of the AOM **Application** and **Acl** interfaces. Only one object for each interface exists in a Arbortext Editor session.

Object	Description
Application	This global object implements the **Application** interface that provides access to all other DOM and AOM objects except for the **Acl** interface.
Acl	This global object implements the **Acl** interface that provides access to ACL (Arbortext Command Language).

VBScript Error Handling

VBScript does not support exceptions, so the DOM and AOM exception classes are not available. All exceptions are mapped to the VBScript `Err` global object. The COM error code values listed in COM Error Handling on page 93 are available using the `Number` property of the **Err** object. The message associated with the exception is available using the `Description` property. For example:

```
On Error Resume Next
doc.insertBefore doc, doc ' this is invalid
If Err.Number <> 0 Then
 Application.alert("Error: " & Err.Number _
 & " Description: " & Err.Description)
 Err.Clear
End if
```

Sample VBScript Code

Sample VBScript code that uses the VBScript AOM interface is included in the *Arbortext-path*`\samples\vbscript` directory. The `readme.txt` file in this directory provides a description of the sample code and instructions for invoking the sample scripts. Examples show how to use the DOM to manipulate the active document and register DOM event handlers. There are two samples, `commdlg.vbs` and `graphic-browser.vbs`, which show how to use COM to launch and communicate with Microsoft Word and Microsoft Excel. The VBScript examples are ported from the corresponding JScript samples of the same name.

III

Programming and Scripting Techniques

10

Overview of Programming and Scripting Techniques

This part of the *Programmer's Reference* contains information on using Arbortext Editor and the AOM to perform basic and advanced operations. Individual chapters include:

- Overview on page 114 — Contains a series of examples demonstrating basic techniques for manipulating documents and content using the DOM and AOM.
- Overview on page 124 — Summarizes the DOM Event Model interfaces and the AOM extended event interfaces supported by Arbortext Editor and the Arbortext Publishing Engine.
- Working with Tables Overview on page 160 — The AOM contains interfaces that provide access to more than 100 Arbortext Editor table functions. This chapter provides several examples that illustrate the basics of inserting and manipulating tables using the interfaces.
- Overview on page 168 — XSL composition refers to Arbortext Editor's ability to transform a document using XSL or XSL-FO stylesheets. This chapter describes XSL composition and its components, and provides an example of calling the composition pipeline for an HTML file composition.
- Line Numbering Overview on page 174 — You can add line numbers to your document, specifying their format using a custom application. This chapter describes the basic line numbering functionality that is available with a Arbortext distributed document type, and detailed instructions for building your own.

11

Basic Document Manipulation Using the DOM and AOM

Overview

This chapter contains a series of brief examples demonstrating basic techniques for manipulating documents and content using the DOM and AOM. The examples cover opening, closing, and saving documents; traversing document trees; inserting text; and locating, selecting, cutting, and pasting content in and between documents.

Most of the sample code in this chapter can be run on the Arbortext XML Docbook sample opened with Arbortext Editor. (Choose **File ▸ New**, check **Sample**, select **Arbortext XML Docbook V4.0**, and click **OK**.) Example code that calls `openDocument` requires access to one or two saved copies of the Arbortext XML Docbook sample.

All of the examples in this chapter are written in JavaScript.

Opening, Closing, and Saving documents

DOM Level 2 does not provide methods to open, save, and close documents. However, the AOM includes methods on the **Application** and **ADocument** interfaces that implement these capabilities.

The **Application** interface `openDocument` method returns a **Document** object that has information about a document or document type and can be used to dynamically update the content, structure, and style of the document

The `openDocument` method takes several optional parameters, including the *flags* parameter, which controls the state in which the document is opened. This parameter is constructed by adding the hex values of the `LoadFlag` enumeration constants. (The symbolic constant names can be used instead with some language bindings.) Refer to Application interface on page 273 for a complete listing and full descriptions of the `LoadFlag` enumeration constants. The following table highlights a selection of these constants.

Name	Hexadecimal value	Description
`OPEN_RDONLY`	`0x0001`	Open the document as read only.
`OPEN_DOCRDWR`	`0x0002`	Open the document for read and write.
`OPEN_NOMSGS`	`0x0020`	Suppress any parser error messages.
`OPEN_EDITINIT`	`0x8000`	Process initialization files upon opening.

In the following code, the *flags* parameter is used to open a document for read and write while suppressing any parser errors:

```
var doc = Application.openDocument("mydocument.xml", (0x0002 + 0x0020))
```

Once a document is opened, it can be manipulated and then saved and closed using methods of the **ADocument** interface (which extends the W3C DOM **Document** interface).

ADocument.save writes the document to disk. The `save` method's **flags** parameter determines the state of the saved document.

ADocument.close frees all resources associated with the **Document** object.

Refer to the examples in the remainder of this chapter for several sample uses of the **Application.openDocument**, **ADocument.save**, and **ADocument.close** methods.

Traversing a Document Using the DOM and AOM

A **Document** object is the tree representation of the document's structure. Like any tree, the document can be traversed several ways.

Traversing and Printing a Document Structure

In this example, as the document is traversed, the tag name and up to the first 60 characters of each node are printed to illustrate the hierarchical structure of the current document.

In addition to demonstrating how to walk a DOM tree, this example also shows how to access the names of nodes (**Node.nodeName**), how to determine a node's type (**Node.nodeType** = text, element, comment, or processing instruction), and how to extract text content from a document (`Node.data`).

```
function printTree(n, elem) {
 if (elem == null) {
 if (n == 0)
 print("document has no element nodes");
 return;
}
 var str = "";
 for (var i = 0; i < n; i++)
 str += " ";
 // show this node
 print(str + elem.tagName + getAttrs(elem));
 str += " ";
 // followed by its children
 for (var child = elem.firstChild; child != null;
 child = child.nextSibling) {
 if (child.nodeType == child.ELEMENT_NODE)
```

```
 printTree(n + 1, child);
 else if (child.nodeType == child.TEXT_NODE) {
 // for text nodes, show the first 60 characters
 // note, concatentation with a null string is used to convert
 // the Java String returned into a JavaScript string.
 var text = child.data + "";
 if (text.length > 60)
 print(str + '"' + text.substr(0, 60) + "...\"");
 else
 print(str + '"' + text + '"');
}
 else if (child.nodeType == child.COMMENT_NODE) {
 var text = "#comment: " + child.data;
 if (text.length > 60)
 text = text.substr(0, 60) + "...";
 print(str + text);
}
 else if (child.nodeType == child.PROCESSING_INSTRUCTION_NODE)
 print(str + "#pi: " + child.target + ' ' + child.data);
 else // all others
 print(str + child.nodeName);
}
}
// start at the root
printTree(0, Application.activeDocument.documentElement);
```

Using getElementsByTagName

In this example, the tree is traversed by calling `getElementsByTagName`. All of the **Document**, **ADocument**, **Element**, and **AElement** interface `getElementsBy`*`Xxx`* methods populate a **NodeList** with nodes in the order encountered in a preorder traversal of the tree. All occurrences of the `<emphasis>` tag have their `role` attribute value changed from `bold` to `italic`, changing all bold text to italic. This is done by iterating over the **NodeList** returned by **getElementsByTagName**, and using `Node.getAttribute` to check the value of each node's `role` attribute, and then using `Node.setAttribute` to change that value to `italic`.

```
var doc = Application.activeDocument;
//get all emphasis tags in the document
var tags = doc.getElementsByTagName("emphasis");
for(i=0; i < tags.length; i++) {
 if(tags.item(i).getAttribute("role") == "bold") {
 tags.item(i).setAttribute("role", "italic")
}
}
```

Using getElementsByAttribute

The previous example could be improved by using the **AElement.getElementsByAttribute** method. (The AOM **AElement** interface extends the W3C DOM **Element** interface.) Doing so will return only those tags from the document that have the `role` attribute set to `bold`. The value on all of the tags can then be changed from `bold` to `italic` without having to test every `<emphasis>` tag in the document.

The `getElementsByAttribute` method takes three arguments: *name*, *value*, and *selector*. If *selector* is set to `1` (one), the search will return all nodes that match both *name* and *value*. If *selector* is set to `0` (zero), all nodes matching *name*, regardless of their value, are returned.

```
var doc = Application.activeDocument;
var tags = doc.getElementsByAttribute("role", "bold", 1);
for (i=0; i < tags.length; i++) {
 tags.item(i).setAttribute("role", "italic");
}
```

Inserting Text

Text can be added at any appropriate place in a document by creating and inserting a new **Text** node. **Document.createTextNode** takes a text string as an argument, and returns a new node (**Text** object) that can be inserted by calling methods such as **Node.appendChild** or **Node.insertBefore** on the desired node.

Inserting Text Using createTextNode

This example appends the line "`Adding new text.`" to the end of the first paragraph in a document

```
var doc = Application.activeDocument;
var paras = doc.getElementsByTagName("para");
//create the new Text Node
var newText = doc.createTextNode(" Adding new text.");
//append it to first paragraph
paras.item(0).appendChild(newText);
```

Inserting Text Containing a Non-Latin Character

To insert a string containing characters such as letters from non-English alphabets, include the Unicode character in the text string. Do not include it as an entity reference.

For example, suppose you are authoring a travel guide and wish to append a paragraph that includes the German word `Gemütlichkeit`. If you include the `ü` as an entity reference, the entity will not be resolved. For example:

```
var newText1 = doc.createTextNode("Austrians are known for their
 Gem&uuml;tlichkeit");
```

The text node will literally contain "Gemütlichkeit". Instead, insert the character as in the following example:

```
var doc = Application.activeDocument;
var paras = doc.getElementsByTagName("para");
var newText = doc.createTextNode(" Austrians are known for their Gemütlichkeit");
paras.item(0).appendChild(newText);
```

Inserting an Entity Reference Using createEntityReference

To insert such characters as an entity references, use **Document.createEntityReference** rather than `createTextNode`. This example produces the same result as the previous example, but uses a character entity to insert the u-umlaut:

```
var doc = Application.activeDocument;
var paras = doc.getElementsByTagName("para");
var newText1 = doc.createTextNode("Austrians are known for their Gem");
var charEnt = doc.createEntityReference("uuml");
var newText2 = doc.createTextNode("tlichkeit");
paras.item(0).appendChild(newText1);
paras.item(0).appendChild(charEnt);
paras.item(0).appendChild(newText2);
```

Using Range to Select and Delete Content

The W3C DOM Range API consists of a single interface, **Range**. This interface exposes the ability to select contiguous portions of a structured document, delineated by specified beginning and end points. The **Range** interface contains methods that allow copying, inserting, or deleting of content, as well as methods for marking the start and end points of the content range.

Deleting Sections of a Document Using a Range

This example illustrates several basic techniques:

- Opening a document using the optional flags parameter (**Application.openDocument**).
- Gathering elements by attribute name and value (`getElementsByAttribute`).
- Prompting for user input (**Application.confirm**).
- Using a range to mark content for deletion and delete it (the `deleteTag` function).
- Handling a **NodeList**.

The result of the code in this example is that the user is prompted with the option to delete all the tags in a document that have a certain profiling attribute.

The `deleteTag` function in the example demonstrates the creation, marking, and use of a **Range** object. First the **Range** must be created (**Document.createRange**). The beginning and end points must then be set (**Range.setStartBefore** and **Range.setEndAfter**). The content in the **Range** is then deleted, and the range is detached.

The call to **Range.detach()** is critical, as this method frees all resources associated with this **Range** object. Any subsequent call on that object would result in an exception being thrown. This method should be called whenever a use of a **Range** object is complete.

```
//Delete the given node (tag and its children and/or contents)
function deleteTag(tag) {
 var range = doc.createRange();
 range.setStartBefore(tag);
 range.setEndAfter(tag);
 range.deleteContents();
 range.detach();
}
//Open the document for writing, while suppressing any parse errors
//OPEN_DOCRDWR(0x0002) - open the document for reading and writing
//OPEN_NMSGS(0x0020) - suppress any parser error messages
var doc = Application.openDocument("sample.xml", (0x0002 | 0x0020));
//Select all tags with the profiling attribute "security" and the value
 "Employee"
var profiles = doc.getElementsByAttribute("security", "Employee", 1);
//Prompt the user to delete the selected tags
var response = Application.confirm("Found " + profiles.length +
 " profiled items.\nOK to delete?", "Confirm Deletion");
//If the user clicked "OK", go ahead and delete them
if(response) {
 while(0 < profiles.length) {
 deleteTag(profiles.item(0));
}
}
```

Notice in this example that in the loop that calls `deleteTag`, it is `item(0)` that is deleted each time. This is because in the W3C DOM **NodeList** specification, **NodeList**s are live. That is, changes in the underlying document object are immediately reflected in the **NodeList**.

For example, if tags had been deleted using the following code, only every other node would have been deleted.

```
for(i = 0; i < profiles.length; i++) {
 deleteTag(profiles.item(i));
}
```

Selecting, Copying, Moving Content

The following examples demonstrate how to copy, cut, and paste content within and between documents.

Cutting and Pasting within a Document

This example swaps the position of the first two chapters in a document. When chapter one is inserted before chapter three, it is the same as a cut and paste; it is not a copy of the node, but the node itself that is being moved.

```
var doc = Application.openDocument("sample1.xml");
//Get the nodes contining chapters one and three from the document
//Chapter three will be the node to insert before
var chapters=doc.getElementsByTagName("chapter");
var chapter1 = chapters.item(0);
var chapter3 = chapters.item(2);
var book = doc.getElementsByTagName("book").item(0);
//chapter1 is the new node, and chapter3 is the reference
book.insertBefore(chapter1,chapter3);
```

Copying and Pasting within a Document

A copy and paste within a document can be done by cloning the contents of chapter one before inserting them before chapter three. In this example, the result will be two copies of chapter one in the document; one before and one after chapter two.

```
var doc = Application.openDocument("sample1.xml");
var chapters=doc.getElementsByTagName("chapter");
var chapter1 = chapters.item(0);
var chapter3 = chapters.item(2);
var book = doc.getElementsByTagName("book").item(0);
var range = doc.createRange();
range.setStartBefore(chapter1);
range.setEndAfter(chapter1);
var clone = range.cloneContents();
book.insertBefore(clone,chapter3);
range.detach();
```

Copying and Pasting between Documents

Content can also be moved between documents using **Document.importNode**. The code in this example results in a copy and paste without the need to clone the region from the first document. This is because **Document.importNode** does not alter or remove content from the original document; it creates a new copy of the source node — in effect, cloning it. This example also demonstrates the use of **ADocument.openDocument**, the use of optional *flags* and *path* parameters on **ADocument.save**, and **ADocument.close**.

```
var doc1 = Application.openDocument("sample1.xml");
var doc2 = Application.openDocument("sample2.xml");
//Get the first chapter from sample1.xml and sample2.xml
var sample1Chapter = doc1.getElementsByTagName("chapter").item(0);
var sample2Chapter = doc2.getElementsByTagName("chapter").item(0);
var book = doc2.getElementsByTagName("book").item(0);
//Import the chapter from sample1.xml into sample2.xml
var newChapter = doc2.importNode(sample1Chapter,true);
//insert the chapter
book.insertBefore(newChapter,sample2Chapter);
//SAVE_NAC_ENTREF(0x0400) - write non-ascii characters as
// character entity references
doc2.save(0x0400, "newSample2.xml");
doc1.close();
doc2.close();
```

To execute a cut and paste between documents, select and delete the contents in the original document after inserting it in the target document.

Inserting Text at the Caret

This example shows how to insert text in the document where the caret is located using the **Range** returned by the `ADocument.insertionPoint` attribute. If the caret is within a text node, the text is inserted into that node. Otherwise, a new text node is inserted before the `insertionPoint` node.

```
var doc = Application.activeDocument;
var caret = doc.insertionPoint;
var node = caret.endContainer;
if (node.nodeType == node.TEXT_NODE)
node.insertData(caret.endOffset, " new text ");
else
caret.insertNode(doc.createTextNode(" new text "));
```

Inserting Markup at the Caret

The **ARange** extension includes the method `insertParsedString`. This method makes it easy to insert strings containing markup (tags and entity references) into a range, including the one that represents the document caret position. The following two examples are equivalent and insert the string “an **emphasized** word” with the second word “**emphasized**” enclosed in `<emphasis>` tags. The first example is implemented using standard DOM methods:

```
var doc = Application.activeDocument;
var caret = doc.insertionPoint;
var node = caret.endContainer;
var parent = node.parentNode;
// does not consider caret offset into text node
parent.insertBefore(doc.createTextNode("an "), node);
var emph = doc.createElement("emphasis");
```

```
emph.appendChild(doc.createTextNode("emphasized"));
parent.insertBefore(emph, node);
parent.insertBefore(doc.createTextNode(" word"), node);
```

The following example uses the `ARange.insertParsedString` method:

```
var doc = Application.activeDocument;
doc.insertionPoint.insertParsedString("an <emphasis>emphasized</> word");
```

12

Events

Overview

Arbortext Editor and the Arbortext Publishing Engine implement the W3C DOM Event Model described in the *Document Object Model (DOM) Level 2 Events Specification* (www.w3.org/TR/DOM-Level-2-Events). The DOM Event Model is a generic event system that provides registration of event handlers, describes the flow of events through a tree structure, and defines contextual information for each event.

Event Interfaces

The following tables summarize the DOM Event Model interfaces and the AOM extended event interfaces supported by Arbortext Editor and the Arbortext Publishing Engine.

W3C Event Interfaces

Interface	Description
DocumentEvent	Implemented by objects that implement the **Document** interface to create user dispatched events.
Event	Provides contextual information for an event handler. The superinterface of more specific event context interfaces.
EventException	Exception thrown by event related methods.
EventListener	Mechanism for handling events.
EventTarget	Implemented by objects that implement the **Node** and **Component** interfaces to allow registration and removal of **EventListener**s and dispatching of events.
MouseEvent	Provides contextual information associated with Mouse events.
MutationEvent	Provides contextual information associated with Mutation events.
UIEvent	Provides contextual information associated with User Interface events.

AOM Event Interfaces

Interface	Description
ADocumentEntityEvent	Provides specific contextual information associated with the `ADocumentEntityEvent` extension.
ADocumentEvent	Provides specific contextual information associated with `document` events.
ActivexEvent	Provides specific contextual information

AOM Event Interfaces (continued)

Interface	Description
	associated with `Activex` events.
AEditEvent	Provides contextual information associated with `EditEvent` events.
AEvent	Extension to the W3C DOM **Event** interface.
ApplicationEvent	Provides specific contextual information associated with `application` events.
CMSObjectEvent	Provides specific contextual information associated with the `CMSObjectEvent` extension.
CMSSessionConstructEvent	Provides specific contextual information associated with the `CMSSessionConstructEvent` extension.
CMSSessionCreateEvent	Provides specific contextual information associated with the `CMSSessionCreateEvent` extension.
CMSSessionFileEvent	Provides specific contextual information associated with the `CMSSessionFileEvent` extension.
CMSSessionBurstEvent	Provides specific contextual information associated with the `CMSSessionBurstEvent` extension.
CMSSessionDisconnectEvent	Provides specific contextual information associated with the `CMSSessionDisconnectEvent` extension.
CMSAdapterConnectEvent	Provides specific contextual information associated with the `CMSAdapterConnectEvent` extension.
CMSAdapterDisconnectEvent	Provides specific contextual information associated with the `CMSAdapterDisconnectEvent` extension.
ControlEvent	Provides specific contextual information associated with `Control` events.
MenuEvent	Provides contextual information associated with Menu events.

AOM Event Interfaces (continued)

Interface	Description
ToolBarEvent	Provides specific contextual information associated with `ToolBar` events.
WindowEvent	Provides contextual information associated with Window events.

Event Modules and Domains

The DOM Level 2 Events specification allows an application to support multiple modules of events. Arbortext Editor and the Arbortext Publishing Engine support all of the DOM Level 2 event modules except `HTMLEvents`. In addition, Arbortext Editor and the Arbortext Publishing Engine add several application-specific event modules and further divide the event modules into the following event domains: `CMSObject`, `CMSSession`, `CMSAdapter`, `Document`, and `Window`.

The `Document` domain includes those events created by the `createEvent` method of the **DocumentEvent** interface and used by the **EventTarget** interface as implemented by the **Node** interface and its subclasses. The `Document` domain includes the DOM Level 2 Event modules `UIEvents`, `MouseEvents`, and `MutationEvents`, as well as the Arbortext-specific `AEditEvent` module. The `AEditEvent` module defines several event types used to notify programmers of important document operations that are not covered by DOM events.

The `Window` domain includes those events created by the `createEvent` method of the **Window** interface and used by the **EventTarget** interface as implemented by the **Component** interface and its subclasses. The `Window` domain includes the `WindowEvents`, `MenuEvents` and `ControlEvents` modules.

The `CMSSession` domain includes those events associated with CMS sessions. The target of all events in this domain is a `CMSSession`. The events in this domain bubble in the following order:

1. `CMSSession`
2. Associated `CMSAdapter`
3. `Application`

An `EventListener` may be established on any of these targets.

The `CMSObject` domain includes those events associated with CMS objects. The target of all events in this domain is a `CMSObject`. The events in this domain bubble in the following order:

1. `CMSObject`
2. Associated `Document` (if any). There may be no associated document, for example, if the object has no associated nodes (such as an object representing a folder in the repository).
3. Associated `CMSSession`
4. Associated `CMSAdapter`
5. `Application`

An `EventListener` may be established on any of these targets.

The `CMSAdapter` domain includes those events associated with CMS adapters. The target of all events in this domain is a `CMSAdapter`. The events in this domain bubble in the following order:

1. `CMSAdapter`
2. `Application`

An `EventListener` may be established on both of these targets.

The **AEvent** interface is the Arbortext extension to the W3C **Event** interface which adds two attributes to determine the domain and module of the event:

- `domain` — returns a constant identifying the event domain
- `moduleType` — returns a constant identifying the event module

The following event modules are supported. The module name listed is the feature string to pass as the *eventType* parameter to the appropriate `createEvent` method.
UIEvents

Events associated with user interaction with a mouse or keyboard.

Domain: Document
MouseEvents

Events associated with mouse input devices.

Domain: Document
MutationEvents

Events associated with actions that modify the structure of the document.

Domain: Document
AEditEvents

Events associated with high level editing operations.

Domain: Document

WindowEvents

Events associated with changes in the state of **Window** objects.

Domain: Window
MenuEvents

Events associated with `MenuItem` objects.

Domain: Window
ControlEvents

Events associated with XUI control objects. These are not currently exposed through the AOM.

Domain: Window
CMSObjectEvent

Events associated with CMS objects.

Domain: CMSObject
CMSSessionConstructEvent

Events associated with construct operations for existing CMS objects.

Domain: CMSSession
CMSSessionCreateEvent

Events associated with creating new CMS objects.

Domain: CMSSession
CMSSessionFileEvent

Events associated with file-related CMS session operations.

Domain: CMSSession
CMSSessionBurstEvent

Events associated with burst-related CMS session operations.

Domain: CMSSession
CMSSessionDisconnectEvent

Events associated with CMS session disconnection operations.

Domain: CMSSession
CMSAdapterConnectEvent

Events associated with CMS adapter connection operations.

Domain: CMSAdapter
CMSAdapterDisconnectEvent

Events associated with CMS adapter disconnection operations.

Domain: CMSAdapter

Note

The **DLMEvent** module supports events associated with the Dynamic Link Manager. It is a Java-only implementation that is documented in the Javadoc available in the Arbortext Editor Help Center.

Application-Dependent Features

The DOM Level 2 Events specification defines the `DOMFocusIn`, `DOMFocusOut`, and `DOMActivate` user interface events, but does not define when they will occur. The specification also allows implementation-dependent treatment of the `DOMSubtreeModified` mutation event. The following table describes when these events occur in Arbortext Editor and the Arbortext Publishing Engine:

Event	Occurrence
`DOMFocusIn`	Two occurrences: • When the cursor of the view that has keyboard input focus moves into an event target. • When the keyboard input focus switches from another view to the current view while the cursor of the current view is inside an event target.
`DOMFocusOut`	Two occurrences: • When the cursor of the view that has keyboard input focus moves out of an event target. • When the keyboard input focus switches from the current view to another view while the cursor of the current view is inside an event target.

Event	Occurrence
`DOMActivate`	When an event target is activated through a mouse double-click. For a XUI document, this event will be dispatched when its corresponding dialog box state changes, such as when a check box is selected, an item of a list box is selected, a push button is pressed, and so on.
`DOMSubtreeModified`	Certain user interface actions like **Insert ► Markup** can result in multiple changes to the document; only a single `DOMSubtreeModified` event will be fired in those cases.

Refer to Event Types on page 135 for a description of each event type.

Notes and Limitations

The following notes and limitations apply to the Arbortext Editor and the Arbortext Publishing Engine implementations of events:

- Be aware that DOM mutation events trigger after the document is loaded and something happens to change the document, not as the document is being read in by Arbortext Editor or the Arbortext Publishing Engine.
- HTML-specific features in the W3C DOM Events specification are not implemented.
- No mutation events are currently fired for undo or redo operations. Instead the `AOMUndo` event type is dispatched.
- SGML-specific document structures such as ignored marked sections are not supported by the Arbortext Editor and the Arbortext Publishing Engine DOM implementation.

Event Handlers

Event handlers are registered in a binding-specific manner. The following sections illustrate the techniques used to implement the **EventListener** interface for each language binding supported by Arbortext Editor and the Arbortext Publishing Engine.

The example (repeated in each binding) shows how to register a mouse click handler (of the `MouseEvents` event module) for the active document. The handler prints a line to the message window showing the element hierarchy in the following form each time the mouse is clicked within the document:

```
(book(chapter(para
```

Java

In Java, it is necessary to cast the **Document** object to call the `addEventListener` method of the **EventTarget** interface. Also, note the event listener parameter is specified using an anonymous inner class.

```
Document doc = Application.getActiveDocument();
((EventTarget)doc).addEventListener("click",
 new EventListener() {
 public void handleEvent(Event event) {
 Node node = (Node)event.getTarget();
 String context = "";
 while (node != null) {
 if (node.getNodeType() == Node.ELEMENT_NODE) {
 context = "(" + node.getNodeName() + context;
}
 node = node.getParentNode();
}
 Application.print(context + "\n");
 event.stopPropagation();
}
}, true);
```

JavaScript

JavaScript uses the LiveConnect feature to connect to Java to create the DOM **EventListener** object to pass to `addEventListener`. The handler object associated with the **EventListener** is declared using object literal syntax.

```
function clickEvent(event)
{
 var node = event.target;
 var context = "";
 while (node != null) {
 if (node.nodeType == node.ELEMENT_NODE) {
 context = "(" + node.nodeName + context;
}
 node = node.parentNode;
}
 Application.print(context + "\n");
 event.stopPropagation();
}
var doc = Application.activeDocument;
// define an object with the required handleEvent method
var o = { handleEvent: clickEvent};
```

```
var listener = Packages.org.w3c.dom.events.EventListener(o);
doc.addEventListener("click", listener, true);
```

JScript

In JScript, the **EventListener** interface is implemented by declaring a constructor of the same name. Note, that because of the way JScript works, the interface constants like **Node.ELEMENT_NODE** are not available. Otherwise, the `clickEvent` function is the same as the in the JavaScript example. The main difference is in how the listener object is created.

```
function EventListener( )
{
 this.handleEvent = clickEvent;
}
function clickEvent(event)
{
 var node = event.target;
 var context = "";
 while (node != null) {
 if (node.nodeType == 1 /*ELEMENT_NODE*/) {
 context = "(" + node.nodeName + context;
}
 node = node.parentNode;
}
 Application.print(context + "\n");
 event.stopPropagation();
}
var doc = Application.activeDocument;
var listener = new EventListener();
doc.addEventListener("click", listener, true);
```

VBScript

In VBScript, the event handler is declared as a class:

```
Class EventListener
 Public Function handleEvent(ByVal evt)
 Dim node
 set node = evt.target
 Dim context
 context = ""
 While Not node Is Nothing
 If node.nodeType = 1 Then
 context = "(" & node.nodeName & context
 End If
 Set node = node.parentNode
 Wend
 Application.print(context)
 Application.print()
 evt.stopPropagation()
```

```
 handleEvent = 0
 End Function
End Class
Dim doc
set doc = Application.activeDocument
Dim listener
set listener = new EventListener
doc.addEventListener "click", listener, true
```

Visual Basic

In Visual Basic, the event handler is created as a listener class with the following code. Note that `Print` is a reserved method name in Visual Basic, so the `Application.Print` method is not available; the VB `Debug.Print` method is used instead.

```
Option Explicit
Implements IDOMEventListener
Private Sub IDOMEventListener_handleEvent _
 (ByVal evt As IDOMEvent)
 Dim node As IDOMNode3
 Set node = evt.target
 Dim context As String
 context = ""
 While Not node Is Nothing
 If node.nodeType = NODE_ELEMENT Then
 context = "(" & node.nodeName & context
 End If
 Set node = node.parentNode
 Wend
 Debug.Print context
 evt.stopPropagation
End Sub
```

Then a Visual Basic form must be created with this code included to register the event listener:

```
Option Explicit
Dim myListener As IDOMEventListener
Dim app As Epic.Application
Dim activeDoc As DOMDocument
Dim target As IDOMEventTarget
Private Sub Form_Load()
 Set myListener = New Listener
 Set app = New Epic.Application
 Set activeDoc = app.ActiveDocument
 Set target = activeDoc
 target.addEventListener "click", myListener, False
End Sub
```

COM C++

Much of the COM C++ example was generated automatically using the **Insert ► New ATL Object** menu in the Microsoft Visual C++ IDE followed by **Implement Interface** on the `CListener` class added by **New ATL Object**. This was edited so both the raw methods and the method wrappers were created by the `#import` statement.

The listener class declaration is:

```
#ifndef __LISTENER_H_
#define __LISTENER_H_
#include "resource.h" // main symbols
#import "epic.exe" raw_native_types, no_namespace, named_guids
class ATL_NO_VTABLE CListener :
 public CComObjectRootEx<CComSingleThreadModel>,
 public IDispatchImpl<IDOMEventListener,
 &IID_IDOMEventListener, &LIBID_Epic>
{
public:
 CListener()
 {
}
DECLARE_NO_REGISTRY()
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CListener)
 COM_INTERFACE_ENTRY(IDispatch)
 COM_INTERFACE_ENTRY(IDOMEventListener)
END_COM_MAP()
public:
 STDMETHOD(raw_handleEvent)(IDOMEvent * evt);
};
#endif //__LISTENER_H_
```

The listener implementation class is:

```
#include "stdafx.h"
#include "Listener.h"
#include <string>
typedef std::basic_string< unsigned short > DOMString;
STDMETHODIMP CListener::raw_handleEvent( IDOMEvent *rawEvent)
{
 IDOMEventPtr pEvent = rawEvent;
 IDOMNode3Ptr pNode = pEvent->target;
 DOMString context;
 while (pNode)
 {
 if (pNode->nodeType == NODE_ELEMENT)
 {
 context.insert(0, pNode->nodeName);
 context.insert(0, L"(");
}
 pNode = pNode->parentNode;
```

```
}
 _Application3Ptr pEpic(__uuidof(Application));
 context += L"\n";
 pEpic->Print(_variant_t(context.c_str()));
 pEvent->stopPropagation();
 return S_OK;
}
```

The method that creates and attaches the listener is:

```
void AttachListener()
{
 CListener *pListener = new CComObject<CListener>;
 IDOMEventListenerPtr pIntfc;
 if (pListener)
 {
 pListener->QueryInterface(IID_IDOMEventListener,
 (void **) &pIntfc);
 _Application3Ptr pEpic(__uuidof(Application));
 IDOMEventTargetPtr pDocTarget;
 pDocTarget = pEpic->ActiveDocument;
 pDocTarget->addEventListener(_bstr_t("click"), pIntfc, true);
}
}
```

Event Types

The following sections define the event types supported by each event module and include information about event bubbling, event cancellation, and specific context information for each event type.

The descriptions of the W3C modules (**UIEvent**, **MouseEvent**, and **MutationEvent**) in the following sections are taken from the Document Object Model (DOM) Level 2 Events Specification (www.w3.org/TR/DOM-Level-2-Events).

UIEvent Module

The W3C **UIEvent** module has the following event types:

DOMFocusIn

The `DOMFocusIn` event occurs when an `EventTarget` receives focus, for instance by a pointing device being moved onto an element or by tabbing navigation to the element. Unlike the HTML event `focus`, `DOMFocusIn` can be applied to any focusable `EventTarget`, not just FORM controls.

- Bubbles: Yes
- Cancelable: No
- Context Info: None

DOMFocusOut

The `DOMFocusOut` event occurs when an `EventTarget` loses focus, for instance by a pointing device being moved out of an element or by tabbing navigation out of the element. Unlike the HTML event `blur`, `DOMFocusOut` can be applied to any focusable `EventTarget` , not just FORM controls.

- Bubbles: Yes
- Cancelable: No
- Context Info: None

DOMActivate

The activate event occurs when an element is activated, for instance, through a mouse click or a key press. A numerical argument is provided to give an indication of the type of activation that occurs: `1` for a simple activation (for example, a simple click or **Enter**), `2` for hyperactivation (for example, a double click or **Shift Enter**).

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *detail* (the numerical value)

MouseEvent Module

The W3C **MouseEvent** module has the following event types:

click

The `click` event occurs when the pointing device button is clicked over an element. A click is defined as a mousedown and mouseup over the same screen location. The sequence of these events is:

```
mousedown
mouseup
click
```

If multiple clicks occur at the same screen location, the sequence repeats with the *detail* attribute incrementing with each repetition. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *button*, *detail*

mousedown

The `mousedown` event occurs when the pointing device button is pressed over an element. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *button*, *detail*

mouseup

The `mouseup` event occurs when the pointing device button is released over an element. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *button*, *detail*

mouseover

The `mouseover` event occurs when the pointing device is moved onto an element. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *relatedTarget* indicates the `EventTarget` the pointing device is exiting.

mousemove

The `mousemove` event occurs when the pointing device is moved while it is over an element. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: No
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*

mouseout

The `mouseout` event occurs when the pointing device is moved away from an element. This event is valid for most elements.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *screenX*, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *relatedTarget* indicates the `EventTarget` the pointing device is entering.

MutationEvent Module

The W3C **MutationEvent** module has the following event types:

DOMSubtreeModified

This is a general event for notification of all changes to the document. It can be used instead of the more specific events listed below. It may be fired after a single modification to the document or, at the implementation's discretion, after multiple changes have occurred. The latter use should generally be used to accommodate multiple changes which occur either simultaneously or in rapid succession. The target of this event is the lowest common parent of the changes which have taken place. This event is dispatched after any other events caused by the mutation have fired.

- Bubbles: Yes
- Cancelable: No
- Context Info: None

DOMNodeInserted

Fired when a node has been added as a child of another node. This event is dispatched after the insertion has taken place. The target of this event is the node being inserted.

- Bubbles: Yes
- Cancelable: No
- Context Info: *relatedNode* holds the parent node

DOMNodeRemoved

Fired when a node is being removed from its parent node. This event is dispatched before the node is removed from the tree. The target of this event is the node being removed.

- Bubbles: Yes
- Cancelable: No
- Context Info: *relatedNode* holds the parent node

DOMNodeRemovedFromDocument

Fired when a node is being removed from a document, either through direct removal of the **Node** or removal of a subtree in which it is contained. This event is dispatched before the removal takes place. The target of this event is the **Node** being removed. If the **Node** is being directly removed the `DOMNodeRemoved` event will fire before the `DOMNodeRemovedFromDocument` event.

- Bubbles: No
- Cancelable: No
- Context Info: None

DOMNodeInsertedIntoDocument

Fired when a node is being inserted into a document, either through direct insertion of the **Node** or insertion of a subtree in which it is contained. This event is dispatched after the insertion has taken place. The target of this event is the node being inserted. If the **Node** is being directly inserted the `DOMNodeInserted` event will fire before the `DOMNodeInsertedIntoDocument` event.

- Bubbles: No
- Cancelable: No
- Context Info: None

DOMAttrModified

Fired after an `Attr` has been modified on a node. The target of this event is the **Node** whose `Attr` changed. The value of *attrChange* indicates whether the `Attr` was modified, added, or removed. The value of *relatedNode* indicates the `Attr` node whose value has been affected. It is expected that string based replacement of an `Attr` value will be viewed as a modification of the `Attr` since its identity does not change. Subsequently replacement of the `Attr` node with a different `Attr` node is viewed as the removal of the first `Attr` node and the addition of the second.

- Bubbles: Yes
- Cancelable: No
- Context Info: *attrName*, *attrChange*, *prevValue*, *newValue*, *relatedNode*

DOMCharacterDataModified

Fired after **CharacterData** within a node has been modified but the node itself has not been inserted or deleted. This event is also triggered by modifications to PI elements. The target of this event is the **CharacterData** node.

- Bubbles: Yes
- Cancelable: No
- Context Info: *prevValue*, *newValue*

AEditEvent Module

The **AEditEvent** extension to the **Event** interface includes the following event types:

AOMCut

The `AOMCut` event occurs before a cut operation is executed. If an event listener doesn't cancel the cut, proper mutation events will be fired after the cut has taken place.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *relatedRange* holds the range that is going to be removed from the document.

AOMCopy

The `AOMCopy` event occurs before the copy operation is executed.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: *relatedRange* holds the range that is going to be copied.

AOMDeleteRegion

The `AOMDeleteRegion` is called before an attempt to delete a contiguous region of a document in an edit window. `AOMDeleteRegion` parallels the `delete_region` ACL callback type, and is dispatched immediately before that callback is invoked. Refer to the `delete_region` documentation for details on when and how this event is fired.

- Bubbles: Yes
- Cancelable: Based on the method by which the content was removed: `true` in cases where *detail* does not contain `0x08`, and `false` if *detail* does contain `0x08`. Refer to the description of `delete_region` for additional details. Calling `preventDefault` if the event is not cancelable will have no effect.
- Context Info: *relatedRange* holds the range containing the content about to be deleted. The *detail* field holds a value identical to the *flags* parameter to the `delete_region` callback.

AOMPaste

The `AOMPaste` event occurs after the paste operation has been executed. Proper mutation events are fired together with the paste event.

- Bubbles: Yes
- Cancelable: No
- Context Info: *relatedRange* holds the range that is newly inserted into the document by the paste operation. *detail* indicates the source of the paste content: 1 for Arbortext Editor, 2 for clipboard.

AOMUndo

The `AOMUndo` event occurs after the undo operation executes. Currently, no mutation events are fired for the undo.

- Bubbles: Yes
- Cancelable: No
- Context Info: *relatedRange* holds the range that is affected by the undo operation. *detail* indicates the source of the undo: 1 for the undo command, 2 for the undo triggered by Arbortext Editor as the result of context errors, 3 for the redo command.

ApplicationEvent Module

The **ApplicationEvent** extension to the **ApplicationEvent** interface includes the following event types:

ApplicationLoad

The `ApplicationLoad` event occurs after Arbortext Editor is initialized and all the startup files in the custom directories have been executed. There is no ACL callback equivalent for this event.

`ApplicationEvent` event listeners need to be registered before Arbortext software is fully loaded. Therefore, a good place to register an `ApplicationLoad` event listener is in a startup file in the custom directory.

- Bubbles: No
- Cancelable: No
- Context Info: None

ApplicationClosing

The `ApplicationClosing` event occurs when the user closes down the Arbortext software. This event type is similar to the ACL session `quit` callback.

This event type is cancelable. If an event listener calls the `preventDefault` method, the closing will be cancelled.

The detail indicates whether the Arbortext software will prompt for document changes or not:

- 0: prompts for any changes.
- 1: saves all modified documents without prompting.
- 2: doesn't prompt for unsaved changes and quits without saving modified documents.

- Bubbles: No
- Cancelable: Yes
- Context Info: detail

ADocumentEvent Module

The **ADocumentEvent** extension to the **Event** interface includes the following event types:

DocumentCreated

The `DocumentCreated` event occurs after a document is constructed and before any document instance startup files are executed. This event type is similar to the ACL document `create` callback. However, the ACL document `create` callback is called after document instance startup files are executed; the `DocumentCreated` event is called before the startup files are executed.

It is impossible to register a `DocumentCreated` event listener in a `Document` object. If the `Document` object exists, the document has already been created. `DocumentCreated` event listeners need to be registered in the `Application` object.

The detail attribute indicates whether the document is empty or not:

- 0: if the document is constructed from a source file.
- 1: if the document is empty.
- Bubbles: Yes
- Cancelable: No
- Context Info: detail

DocumentClosed

The `DocumentClosed` event occurs when a document is destroyed. This event is similar to the ACL document `destroy` callback.

- Bubbles: Yes
- Cancelable: No
- Context Info: None

DocumentLoad

The `DocumentLoad` event occurs when a document is loaded into a window frame and all document instance startup files have been executed. This event is similar to ACL `editfilehook` hook.

When a new window frame is launched, a `DocumentLoad` event will be dispatched for the document displayed in the new window frame.

A window frame can have more than one view. A `DocumentLoad` event will only be dispatched if a document is loaded into a window frame and the document does not already have a view in that window frame.

A document can be loaded into two or more different window frames. A `DocumentLoad` event will be dispatched when a document is loaded into a window frame event if the same document is already displayed in another window frame.

`relatedWindow` specifies the window frame into which the document is loaded.

- Bubbles: Yes
- Cancelable: No
- Context Info: `relatedWindow`

DocumentUnload

The `DocumentUnload` event occurs when a document is unloaded from a window frame. There is no ACL callback equivalent for this event.

A `DocumentUnload` event will only be dispatched if a document is unloaded from a window frame and the document does not have another view in that window frame.

`relatedWindow` specifies the window frame from which the document is unloaded. `relatedWindow` is not set if the window frame is also being destroyed.

- Bubbles: Yes
- Cancelable: No
- Context Info: `relatedWindow` if the window frame still exists. Otherwise, null.

DocumentSaving

The `DocumentSaving` event occurs when the user saves a document. This event type covers ACL document `save` and `saveas` callbacks. The `write` command does not cause any ACL callbacks to be called, but it triggers the `DocumentSaving` event.

This event type is cancelable. If an event listener calls the `preventDefault` method, the save will be canceled. The user can cancel the save and call the `ADocument Save` method in the event listener to save the document. This is useful when some actions need to be done before or after the save.

The `targetURI` specifies the path the document is saved in. The `targetEncoding` specifies the encoding the document is saved in.

The `detail` indicates the command that caused the event:

- 0: if the event is caused by a `save` command.
- 1: if the event is caused by a `saveas` command.
- 2: if the event is caused by a `write` command.
- Bubbles: Yes
- Cancelable: No
- Context Info: `targetURI`, `targetEncoding`, detail

ADocumentEntityEvent Module

The **ADocumentEntityEvent** extension to the **Event** interface includes the following event type:

EntityDeclConflict

The `EntityDeclConflict` event occurs when an entity declaration in an internal subset conflicts with one in an external subset (usually a DTD) or with one in a referencing parent document. This event type is similar to the `entitydeclconflict` ACL callback.

The following module properties provide the context information for this event:

`object`

The `CMSObject` in which the declaration was found.

`relatedDocument`

The `Document` in which the declaration was found.

`relatedNode`

DOM `Entity` containing information about the entity declaration.

To avoid the default behavior (which is to ignore the conflicting entity declaration), the event handler must set the `result` property to specify an alternative entity name as well as call `preventDefault`. Even if `result` is set and `preventDefault`is called, the conflicting declaration will still be ignored if any of the following are true:

- `result` was set to a blank or null string.
- `result` was set to a name which conflicts with an already existing entity.
- `result` was set to an invalid entity name.

Note

Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `object`, `relatedDocument`, `relatedNode`

WindowEvent Module

The **WindowEvent** module has the following event types:

WindowCreated

The `WindowCreated` event occurs when a window is created. This event is similar to the ACL window `create` callback.

It is impossible to register a `WindowCreated` event listener in a `Window` object; if the `Window` object exists, the window has already been created. `WindowCreated` event listeners need to be registered in the `Application` object.

The `WindowCreated` event type bubbles to the `Application` object.

- Bubbles: Yes
- Cancelable: No
- Context Info: None

WindowLoad

This event type is triggered when a window is opened at the first time.

The `WindowLoad` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

WindowClosing

This event type is triggered when the user requests a window be closed through the system menu, through a close button on a window's title bar, or through a platform-defined keystroke, such as **Alt-F4** on Windows.

The `WindowClosing` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: Yes
- Context Info: None

WindowClosed

This event type is triggered after a window is disposed.

The `WindowClosed` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

WindowActivated

This event type is triggered when a window is activated, that is, when it is given the keyboard focus and becomes the active window.

The `WindowActivated` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

WindowDeactivated

This event type is triggered when a window ceases to be the active window.

The `WindowDeactivated` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

WindowMinimized

This event type is triggered when the user minimizes a window.

The `WindowMinimized` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

WindowRestored

This event type is triggered when a window is restored from a minimized state to its previous displayed window size and position.

The `WindowRestored` event type bubbles to the `Application` object.

- Bubbles: No
- Cancelable: No
- Context Info: None

MenuEvent Module

The **MenuEvent** module has the following event types:

MenuPost

This event is dispatched before a menu item is displayed. The target of the event is the `MenuItem` being displayed. This event provides an opportunity for application programmers to disable or enable the menu item based on the nature of the current document or current cursor location.

- Bubbles: No
- Cancelable: No
- Context Info: None

MenuSelected

This event is dispatched when a menu item is selected. The target of the event is the `MenuItem` being selected. The default action of this event is to execute the ACL commands attached to the menu item. If the `preventDefault` method is called, the default action will not occur.

- Bubbles: No
- Cancelable: Yes
- Context Info: None

CMSObjectEvent Module

The **CMSObjectEvent** module has the following event types:

CMSObjectPreCheckin

This event occurs before an object is checked in and before any supporting calls have been made. This event is similar to the `precheckin` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the checkin will be canceled. The event handler can perform a customized checkin itself and then cancel the default checkin by calling `preventDefault` and setting `result` to the result of the checkin.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: None

CMSObjectCheckin

This event occurs before an object is checked in and after some transactional and bursting calls have been made. Specifically, if the adapter supports transactions, a transaction will have been already started, and if the adapter specifies that objects should be burst on checkin then this bursting will already have occurred. If bursting modified the object contents, the object will also have been saved back to the repository.

This event is similar to the `checkin` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the checkin will be canceled. In this case, the pending transaction (if supported) will be rolled back.

The event handler can perform a customized checkin itself and then cancel the default checkin by calling `preventDefault` and setting `result` to the result of the checkin. In this case, the specified result will be used and the transaction will be committed.

Note

Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: None

CMSObjectPostCheckin

This event occurs after an object has been checked in. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:
`result`

- Represents the object that has been checked in.
  - Bubbles: Yes
  - Cancelable: No

- Context Info: `result`

CMSObjectCheckout

This event occurs before an object has been checked out. This event is similar to the `lock` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the checkout will be canceled. The event handler can perform a customized checkout itself and then cancel the default checkout by calling `preventDefault` and setting `result` to the result of the checkout.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module property provides the context information for this event:
`flags`

Defined according to the flags parameter of the `CMSObject.checkout` method.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `flags`

CMSObjectPostCheckout

This event occurs after an object has been checked out. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:
`result`

Represents the object that has been checked out.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `result`

CMSObjectCancelCheckout

This event occurs before an object's checkout has been canceled. This event is similar to the `unlock` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the checkout will remain. The event handler can perform a customized cancellation of the checkout itself and then cancel the default behavior by calling `preventDefault` and setting `result` to the result of the canceled checkout.

Note

Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: None

CMSObjectPostCancelCheckout

This event occurs after an object's checkout has been canceled. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:

`result`

Represents the object whose checkout has been canceled.

- Bubbles: Yes
- Cancelable: No
- Context Info: `result`

CMSObjectSave

This event occurs before an object has been saved. This event is similar to the `save` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the save will be canceled. The event handler can perform a customized save itself and then cancel the default save by calling `preventDefault` and setting `result` to the result of the save.

Note

Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module properties provide the context information for this event:

`flags`

Defined according to the flags parameter of the `CMSObject.save` method.

`start`

Along with `end`, represents the content being saved.

`end`

Along with `start`, represents the content being saved.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `flags`, `start`, `end`

CMSObjectPostSave

This event occurs after an object has been saved. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:

`result`

Represents the object that has been saved.

- Bubbles: Yes
- Cancelable: No
- Context Info: `result`

CMSSessionConstructEvent Module

The **CMSSessionConstructEvent** module has the following event types:

CMSSessionConstructObject

This event occurs before an in-memory `CMSObject` has been constructed corresponding to a repository object. This event is similar to the `construct` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the object will not be constructed. The event handler can perform a customized construction itself and then cancel the default construction by calling `preventDefault` and setting `result` to the result of the construction.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module properties provide the context information for this event:
`logicalId`

Represents the object in the repository.
`relatedNode`

Represents `null` or a `Document` used for contextual information during the construction.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `logicalId`, `relatedNode`

CMSSessionPostConstructObject

This event occurs after an object has been constructed. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:
`result`

Represents the `CMSObject` which has been constructed.

- Bubbles: Yes
- Cancelable: No
- Context Info: `result`

CMSSessionCreateEvent Module

The **CMSSessionCreateEvent** module has the following event types:

CMSSessionCreateNewObject

This event occurs before a new repository object is created. This event is similar to the `create` ACL callback associated with the `sess_add_callback` function. Modifying the `name` or `folderLogicalId` arguments is functionally equivalent to the ACL object naming and object location hooks specified in burst configuration files.

This event type is cancelable. If an event listener calls the `preventDefault` method, the object will not be created. The event handler can perform a customized creation itself and then cancel the default creation by calling `preventDefault` and setting `result` to the result of the construction.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module properties provide the context information for this event:

`name`

Represents the name of the object being created.

`type`

Represents an adapter-specific object type string.

`folderLogicalId`

Represents the parent folder for the new object.

`flags`

Same as the flags parameter of the `CMSSession.createNewObject` method.

`start`

Along with `end`, represents the content of the new object.

`end`

Along with `start`, represents the content of the new object.

`version`

Represents an adapter-specific version for the new object.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `name`, `type`, `folderLogicalId`, `flags`, `start`, `end`, `version`

CMSSessionPostCreateNewObject

This event occurs after an object has been created. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:

`result`

Represents the `CMSObject` which has been constructed.

- Bubbles: Yes
- Cancelable: No
- Context Info: `result`

CMSSessionFileEvent Module

The **CMSSessionFileEvent** module has the following event types:

CMSSessionGetFile

This event occurs before the content of a repository object is downloaded to a local disk file. This event is similar to the `getfile` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the object will not be downloaded. The event handler can perform a customized download itself and then cancel the default download by calling `preventDefault` and setting `result` to specify a local disk file containing the object content.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module properties provide the context information for this event:

`logicalId`

Represents the object whose content is desired.

`notation`

Represents an adapter-specific format specification.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `logicalId`, `notation`

CMSSessionPostGetFile

This event occurs after an object's content has been downloaded. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module properties provide the context information for this event:

`logicalId`

Represents the object whose content is desired.

`notation`

Represents an adapter-specific format specification.

`localPath`

Represents the local disk file containing the object content.

- Bubbles: Yes
- Cancelable: No
- Context Info: `logicalId`, `notation`, `localPath`

CMSSessionPutFile

This event occurs before a new repository object is created from the contents of a local file or other resource. This event is similar to the `putfile` ACL callback associated with the `sess_add_callback` function.

This event type is cancelable. If an event listener calls the `preventDefault` method, the object will not be created. The event handler can perform a customized creation itself and then cancel the default creation by calling `preventDefault` and setting `result` to specify the logical id of the new object.

> **Note**
>
> Setting `result` without calling `preventDefault` will cause the result to be ignored and the default processing to proceed.

The following module properties provide the context information for this event:

`localPath`

Represents the local resource whose content will go into the new object.

`notation`

Represents an adapter-specific format specification.

`objectName`

Represents the name of the new object.

`folderLogicalId`

Represents the parent folder of the new object.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `localPath`, `notation`, `objectName`, `folderLogicalId`

CMSSessionPostPutFile

This event occurs after the new object has been created with the contents of a local resource. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module properties provide the context information for this event:

`localPath`

Represents the local resource whose content went into the new object.

`notation`

Represents an adapter-specific format specification.

`logicalId`

Represents the logical id of the new object.

- Bubbles: Yes
- Cancelable: No
- Context Info: `localPath`, `notation`, `logicalId`

CMSSessionBurstEvent Module

The **CMSSessionBurstEvent** module has the following event types:

CMSSessionBurstDocument

This event occurs before a document is burst into the repository.There is no equivalent ACL hook for this event.

The event handler's ability to assign new values to the `topLevelName` and `folderLogicalId` properties can replace object location and naming rule hooks, which are implemented as inline ACL code in a burst configuration file.

This event type is cancelable. If an event listener calls the `preventDefault` method, the burst will be canceled. In this case, the pending transaction (if supported) will be rolled back.

The following module properties provide the context information for this event:

`canOverride`

Represents whether the event handler is allowed to override the `topLevelName` and `folderLogicalId` properties. If `canOverride` is false, then any changes to these properties will have no effect. If `canOverride` is true, then the event handler can set new values for these properties if desired.

`topLevelName`

Represents the name of the top-level object which will result from bursting the document. This may be `null` or empty which means the name will be auto-generated according to the bursting rules for this adapter. The event handler can override this value if `canOverride` is true.

`folderLogicalId`

Represents the repository folder which will hold the top-level object which will result from bursting the document. This may be `null` or empty which means the folder will be chosen according to the bursting rules for this adapter. The event handler can override this value if `canOverride` is true.

`document`

Represents the document being burst.

`flags`

Same as the flags parameter to the `CMSSession.burstDocument` method.

- Bubbles: Yes
- Cancelable: Yes
- Context Info: `canOverride`, `topLevelName`, `folderLogicalId`, `document`, `flags`

CMSSessionPostBurstDocument

This event occurs after a document has been burst. As such, it is not cancelable. There is no equivalent ACL hook for this event.

The following module property provides the context information for this event:

`document`

Represents the document which has been burst.

- Bubbles: Yes
- Cancelable: No
- Context Info: `document`

CMSSessionDisconnectEvent Module

The **CMSSessionDisconnectEvent** module has the following event type:

CMSSessionPreDisconnect

This event occurs before a a user logs off the repository. There is no equivalent ACL hook for this event. This event type is not cancelable.

The following module property provides the context information for this event:

`currentUser`

Specifies the current CMS user name. This will normally match the `loginId` parameter to the `CMSAdapter.connect` method which established this session.

- Bubbles: Yes
- Cancelable: No
- Context Info: `currentUser`

CMSAdapterConnectEvent Module

The **CMSAdapterConnectEvent** module has the following event type:

CMSAdapterPreConnect

This event occurs before the adapter's `connect` method is invoked. An associated event handler can ensure any resource dependencies are satisfied.

This event type is cancelable. If an event listener calls the `preventDefault` method, the adapter's `connect` method will not be called.

No context information is provided for this event.

- Bubbles: Yes
- Cancelable: Yes

CMSAdapterDisconnectEvent Module

The **CMSAdapterDisconnectEvent** module has the following event type:

CMSAdapterPostDisconnect

This event occurs after a session has successfully logged off the CMS, and as such is not cancelable. An associated event handler can be used to clean up any resource dependencies. The event `CMSSessionPreDisconnect` occurs before the user logs off the repository. When `CMSAdapterPostDisconnect` occurs, the session is invalid, and thus appears in a separate interface.

The following module property provides the context information for this event:
`currentUser`

Specifies the current CMS user name. This will normally match the `loginId` parameter to the `CMSAdapter.connect` method which established this session.

- Bubbles: Yes
- Cancelable: No
- Context Info: `currentUser`

13

Working with Tables

Working with Tables Overview

The AOM contains interfaces that provide access to more than 100 Arbortext Editor table functions. With these interfaces, you can programmatically create and modify tables in any Arbortext Editor document using Java, JavaScript, VB, or VBScript. The entire Arbortext Editor table object model is exposed through the following set of interfaces:

Interface	Description
TableCell	A cell in a table.
TableColumn	A column in a table.
TableException	The `Exception` type thrown when an error is encountered.
TableGrid	In the Oasis Exchange Table model, a table consists of one or more grids, each of which can have a unique number of rows and columns. In the HTML and Arbortext table models, the grid is the sum of all the table rows and columns. This interface allows operation on those grids.
TableMulticell	A rectangular array of spanned cells in a table.
TableObject	The superinterface for **TableCell**, **TableColumn**, **TableGrid**, **TableObjectStore**, **TableRow**, **TableRule**, **TableSet**, and **TableTilePlex**.
TableObjectStore	A collection of **TableObject**s.
TableRectangle	A rectangle of contiguous cells.
TableRow	A row in a table.
TableRule	A rule in a table.
TableSet	A collection of one or more **TableGrid**s.
TableTilePlex	A collection representing a table selection.

The following three code samples illustrate the basics of inserting and manipulating tables using these interfaces. The sample code is in JavaScript. The code will also work using the Microsoft JScript Engine with the noted modifications.

Example: Inserting and Modifying a Table

This example uses the function `addTable` to perform the following actions:

- Insert a six-row five-column table into the first paragraph of a Arbortext XML Docbook template.
- Span cells 1-2 and 3-5 of the first row and add text to the spanned cells.
- Convert the first row to a header row.
- Turn off rules for the entire table.

The function `appendText` is a utility function for adding text to a cell.

To run this sample code:

1. Copy `addTable` and `appendText` to a file named `addtable.js` in *`Arbortext-path`*`\custom\scripts`.
2. Start Arbortext Editor, open a Arbortext XML Docbook template, and enter the following commands at the Arbortext Editor command line:
   ```
   source addtable.js
   js addtable
   ```

```
//-------------------------------------------------------------------
// Function: appendText
//
// Description: A utility function called by addTable.
// Adds text to a cell
//
// Parameters: cell: the target for the added text
// text: the text to be added
//
//-------------------------------------------------------------------
function appendText(cell, text)
{
 var cellRange = cell.contents;
 cellRange.collapse( false );
 var textNode = cell.document.createTextNode(text);
 cellRange.insertNode(textNode);
}
//-------------------------------------------------------------------
// Function: addTable
//
// Description: Add a table to the first para in a document
//
// Parameters: NONE
//
//-------------------------------------------------------------------
function addTable(){
 var doc = Application.activeDocument;
 var para = doc.getElementsByTagName("para").item(0);
```

```
 try{
 var set = para.insertTable("OASIS Exchange", "table", 5, 6, null);
}
 catch(e){Application.alert("Exception " + e.code() +
 " caught in insertTable");
 return 0;}
 var grid = set.grids.item(0);
 var firstRow = grid.row(1);
 // Span cells 1-2 and 3-5
 firstRow.cell(1).span(firstRow.cell(2));
 firstRow.cell(3).span(firstRow.cell(5));
 appendText(firstRow.cell(1), "Cells 1 and 2");
 appendText(firstRow.cell(3), "Cells 3-5");
 // Change first row to a header row
 firstRow.setAttribute("header_level",1);
 //turn off the table rules
 var rules = grid.rules;
 for (i = 0; i < rules.length; i++) {
 rules.item(i).setAttribute("style", "blank");
}
}//end of addTable
```

Example: Inserting a Column Based on the Current Selection

This example uses the function `tbl_insert_column` to insert a column to the left of the current selection. If the selection is invalid, that is, it is discontiguous or not a rectangle, a message is displayed in a dialog box and `tbl_insert_column` returns zero.

To run this sample code:

1. Copy the `tbl_insert_column` code to a file named `insertcol.js` in *`Arbortext-path`*`\custom\scripts`.
2. Start Arbortext Editor, open a Arbortext XML Docbook template, insert a 5x5 table, and enter the following command at the Arbortext Editor command line:
   `source insertcol.js`
3. Select a portion of the table.
4. Enter the following command at the Arbortext Editor command line:
   `js tbl_insert_column()`

```
//-----------------------------------------------------------------
// Function: tbl_insert_column
//
// Description:
// Inserts one or more columns into a document
//
```

```
// Parameter:
// insertLeft: if true (nonzero), adds columns to the left of
// the target
//
// Returns:
// 0 if the insert failed, 1 if it succeeded
//
//-------------------------------------------------------------------
function tbl_insert_column(insertLeft)
{
 if(insertLeft == undefined){insertLeft = 0;}

 var doc = Application.activeDocument;
 //Check to see that there's either a table selection, or that the
 //cursor is in a table cell.
 //To see of a cursor is in a cell:
 //get the range that is the cell containing the cursor
 //get the cell node
 //get the cell containing the caret
 if((doc.selectionType != doc.TABLE_SELECTION) &&
 ((cell = doc.insertionPoint.endContainer.enclosingCell) == null)){
 Application.alert("No table object is selected");
 return 0;
}
 //get the table selection from the active document
 var tilePlex = doc.tableSelection;

 //if the selection is empty, i.e., just a cursor in a cell,
 //add that cell to the tableTilePlex to create a 1x1 rectangle
 if(tilePlex.empty){
 tilePlex.addObject(cell);
}

 //ensure table selection will accept inserted columns
 if(!tilePlex.modifiable){
 Application.alert("table cannot be modified");
 return 0;
}

 //ensure table selection is contiguous and does not cross
 //grid boundaries
 var validRectangle = tilePlex.pasteRectangle;
 if(validRectangle == null){
 Application.alert("The table selection is discontiguous or crosses
 grid boundaries");
 return 0;
}

 //At this point, the selection is valid and can be modified, add the
 //columns to the grid.
 //A new column is added for each one that the user has selected.
```

```
 var newGrid = validRectangle.lowerLeft.grid;
 for(i = 0; i < validRectangle.width; i++){
 try{
 if(insertLeft){
 newGrid.addColumn(validRectangle.lowerLeft.column);
}
 else{
 newGrid.addColumn(validRectangle.upperRight.column.columnRight);
}
}
 catch(e){Application.alert("Column insertion failed because " + e.code);}
}

 //success
 return 1;
}//end of tbl_insert_column
```

To implement the previous example using JScript, change the line:

```
 if((doc.selectionType != doc.TABLE_SELECTION) &&
```

to be:

```
 if((doc.selectionType != 2) &&
```

Example: Identifying a Document Type's Table Model Support

This example uses the function `tableModelInfo` to print all the available information on the current document type's supported table model(s) to the Arbortext Editor message window.

To run this sample code:

1. Copy the `tableModelInfo` code to a file named `tableinfo.js` in *`Arbortext-path`*`\custom\scripts`.
2. Start Arbortext Editor, open a Arbortext XML Docbook or an XHTML v1.0 template, and enter the following commands at the Arbortext Editor command line:

   ```
   source tableinfo.js
   js tableModelInfo
   ```

```
//-----------------------------------------------------------------
// Function: tableModelInfo
// Description: Print all information about the current table models
// Parameters: NONE
//-----------------------------------------------------------------
function tableModelInfo()
{
 var docType = Application.activeDocument.doctype;
 var tblModels = docType.tableModels;
```

```
 Application.alert("Table model information for the " +
 docType + "doctype");
 Application.alert("Number of table models = " + tblModels.length);
 for (var i = 0; i < tblModels.length; i++) {
 Application.print(" [" + i + "] = '" + tblModels.item(i) + "'");
 Application.print(" Supports multiple grids = " +
 docType.tableModelSupport(tblModels.item(i), "multiplegrids"));
 Application.print(" Supports headers = " +
 docType.tableModelSupport(tblModels.item(i), "HeaderRows"));
 Application.print(" Supports footers = " +
 docType.tableModelSupport(tblModels.item(i), "FooterRows"));
 var wrappers = docType.tableModelWrappers(tblModels.item(i));
 Application.print(" Number of wrapper tags = " + wrappers.length);
 for (var j = 0; j < wrappers.length; j++) {
 Application.print(" [" + j + "] = '" + wrappers.item(j) + "'");
}
 var tags = docType.tableModelTags(tblModels.item(i));
 Application.print(" Number of table model tags = " + tags.length);
 for (j = 0; j < tags.length; j++) {
 Application.print(" [" + j + "] = '" + tags.item(j) + "'");
}
}
}//end of tableModelInfo
```

14

Working with XSL Composition

Overview

XSL composition refers to Arbortext Editor's ability to transform a document using XSL or XSL-FO stylesheets. XSL composition is defined by a composer. A composer is a configurable processor that transforms a document by passing it through one or more SAX filters in a filter pipeline.

Filters are classes written in Java that process an input data stream into an output data stream. The data to be processed is represented as a series of SAX events.

A pipeline is a sequence of filters. Each filter takes inputs and produces outputs that get passed to the next filter in the pipeline. A running pipeline is a closed system with a well-defined input (the source) and a well-defined output (the sink).

You specify the parameters for a composer in a composer configuration file (`.ccf`). The `.ccf` file defines composer parameters, including filter resources and the processing sequence.

You can create and edit `.ccf` files using the DCF Editor in Arbortext Architect (**Edit ► CCF**). Several `.ccf` files are distributed with Arbortext Editor. They are located at *`Arbortext-path`*`\composer`.

Related AOM Interfaces and Methods

You can use the following AOM interfaces and methods to obtain information about a composer:

Interface	Description
Application	The `createComposer` method returns a composer object.
Composer	The `getDefaultParameters` method returns a property map of composer parameters in the pipeline definition. The `runComposer method` runs a pipeline associated with the composer object. The `getParameterLabel` method returns the label for the given pipeline parameter. The `getParamDocumentation` method returns the documentation for the given pipeline parameter. The `getParamType` method returns the type for the given pipeline parameter. The `getParamEnumerationValue` method returns all possible values for the enumeration as a string list. The `isParamRequired` method determines if the given parameter is required.

Example: Composing an HTML File

The following example calls the composition pipeline for an HTML file composition.

```
/*
 * ComposerExample is an example of calling the Composition pipeline
 * using the AOM Composer. In this example, an XML document is
 * composed into an HTML file. The source document can exist in one
 * of 2 places:
 * - in Arbortext.
 * - in a file.
 * The Composition uses the htmlfile pipeline defined in htmlfile.ccf
 * in the composer directory.
```

```
 */
import com.arbortext.epic.*;
import org.w3c.dom.*;
import java.io.File;
public class ComposerExample {
 /**
 * Used internally to access the composer configuration file.
 */
 private static final String HTMLFILE_CCF =
 File.separator + "composer" + File.separator + "htmlfile.ccf";
 /**
 * Used internally to access the entity substitution file.
 */
 private static final String HTMLENTSUBFILE =
 File.separator + "composer" + File.separator + "htmlEntSub.xml";
 /**
 * Produces HTML from an in-memory XML file and an XSL stylesheet.
 *
 * @param docId Id of document to process.
 *
 * @param stylesheet Fully-pathed XSL stylesheet.
 *
 * @param outputFile Fully-pathed HTML output filename.
 */
 public static void composeToHtmlFromDoc(int docId, String stylesheet,
 String outputFile) {
 boolean calledStartJob = false;
 try {
 String installPath = Acl.eval("main::aptpath");
 //Create the Composer object for the HTML composition process.
 Composer composer = Application.createComposer(installPath +
 HTMLFILE_CCF);
 PropertyMap params = Application.createPropertyMap();

 //Set up the parameters .
 params.putString("stylesheet", stylesheet);
 params.putString("document", Integer.toString(docId));
 //the entity substitution file for HTML
 params.putString("html.entSubFname", installPath + HTMLENTSUBFILE);
 params.putString("outputFile", outputFile);
 //The following sets up the directory where any graphics would
 //be placed and the associated href in the HTML document.
 params.putString("graphicsHref", (new File(outputFile)).getName()
 + ".graphics/");
 params.putString("graphicsPath", outputFile + ".graphics/");
 // Let the composer know we are using an XSL stylesheet as opposed
```

```
 // to a FOSI ("fosi").
 params.putString("stylesheetType", "xsl");

 //The Acl.* methods perform some initialization that needs to
 //happen for the Composer Log.
 Acl.execute("require _composerlog");
 Acl.execute("require _eventlog");
 //The start_job method MUST be called before the composition process
 //is run.
 Acl.func("_composerlog::start_job", "ComposerExample");
 calledStartJob = true;
 //Set the log level to info.
 String SEVERITY_INFO = Acl.func("eval", "_eventlog::SEVERITY_INFO");
 Acl.func("_composerlog::set_log_severity", SEVERITY_INFO);
 //runPipeline returns a boolean indicating success or failure.
 if (composer.runPipeline(params)) {
 Acl.func("_composerlog::add_record", SEVERITY_INFO, "Success.");
}
 else {
 // Error information will have been placed into the Composer Log.
 Acl.func("_composerlog::add_record", SEVERITY_INFO, "Failure.");
}
}
 catch (AclException ex) {
 // Unexpected.
 System.err.println("ACLException in composeToHtmlFromDoc: " + ex);
 ex.printStackTrace(System.err);
}
 catch (AOMException aomex) {
 // Unexpected.
 System.err.println("AOMException in composeToHtmlFromDoc: " + aomex);
 aomex.printStackTrace(System.err);
}
 finally {
 //Cleanup code to tell the ComposerLog that processing is over.
 // This MUST be called if start_job was called.
 if (calledStartJob) {
 Acl.func("_composerlog::end_job");
}
}
}
 /**
 * Produces HTML from an on-disk XML file and an XSL stylesheet.
 *
 * @param inputFile Fully-pathed XML filename.
 *
```

```
 * @param stylesheet Fully-pathed XSL stylesheet.
 *
 * @param outputFile Fully-pathed HTML output filename.
 */
 public static void composeToHtmlFromFile(String inputFile,
 String stylesheet, String outputFile) {
 ADocument doc = null;
 try {
 doc = (ADocument) Application.openDocument(inputFile);
 composeToHtmlFromDoc(doc.getAclId(), stylesheet, outputFile);
}
 catch (AOMException aomex) {
 System.err.println("AOMException in composeToHtmlFromFile: " + aomex);
 aomex.printStackTrace(System.err);
}
 finally {
 if (doc != null) {
 doc.close();
}
}
}
}
```

15

Line Numbering in Arbortext Editor and Arbortext Publishing Engine

Line Numbering Overview

Arbortext Editor and the Arbortext Publishing Engine provide a framework for building a custom application to add line numbers to XML documents. Line numbers and page numbers can be displayed in the Editor view as well as composed print output.

Applying Line Numbers

Arbortext Editor and Arbortext Publishing Engine provide a framework for building a custom application to add line numbers to XML documents. Line numbers and page numbers can be displayed in the Edit window as well as composed print output.

> **Note**
>
> Using line numbering with the Advanced Preference `deepcontentsplitting` set to `on` may produce unexpected results. It is recommended that you do not use line numbering with `deepcontentsplitting` enabled.

Line Numbering Sample Application

A sample line numbering application can be found in the `samples\linenumbering` folder in your installation directory. Use the following procedure to view an example of line numbering using this sample application. You'll need to have either Arbortext Styler or Print Composer installed and licensed to perform the following procedure:

To Apply Line Numbers to a Sample Document:

1. Choose **File ► New**, select the **Sample** check box, and choose **Arbortext Simplified XML DocBook Article**.
2. At the Arbortext Editor command line, type: `linenum`Line numbers will appear directly to the left of each line in your document.
3. Choose **File ► Print Preview** and use the `asdocbook.style` stylesheet to view the line numbers in a composed document.
4. To remove line numbers from your document, on the Arbortext Editor command line, type: `layout::clear()`

Line Numbering Namespace

The line numbering namespace and associated markup (`atipl` tags) are described on the PTC Arbortext namespace web site at http://www.arbortext.com/namespace/index-of-arbortext-namespaces.html.

Line Numbering Limitations

- Line numbers cannot be added to lines that consist entirely of generated text (for example, a table of contents or index).
- FOSI stylesheets must be used. Line numbering is not supported with XSL-FO stylesheets.
- The same FOSI must be used to apply and view the line numbers.
- Performance on large documents will be slow and memory intensive.
- Changes made outside of Arbortext Editor or Arbortext Publishing Engine may corrupt line and page markers.
- Change tracking records must be either accepted or rejected before line numbering is applied.
- Line numbers can only be displayed on the left side of the Edit window. However, line numbers can be set to appear on either side of a composed print document.
- There is no support for languages without spaces between words (for example, Chinese, Japanese, and Korean).
- Line numbering is only intended to work with XML documents.
- Line numbering is not supported when using composition pipeline formatting (for example, line numbers cannot be applied to profiled documents).
- Line numbering cannot be applied to documents that contain file entities that are referenced multiple times in a single document. Unexpected behavior may result.
- Rules and leaders are ignored. Adjacent line breaks may not be marked up correctly.

The following limitations apply to the sample application, but are not necessarily limitations of the Arbortext Editor line numbering capability

- Only single column output is supported.
- Tables are accommodated, but not algroups.
- Vertical spanning of cells is not supported.
- Only top justified text in tables is supported.

Contact PTC Inc. consulting services for help developing your customized line numbering application.

Building a Basic Line Numbering Application

Use the following procedure to build a rudimentary application that will add line numbers to an XML document. You can use the sample application code found in the `linenum.acl` file in `samples\linenumbering` folder of your installation directory as a starting point or build the application entirely from scratch.

Note

If you are editing SGML documents, remember to recompile your document type to add the line numbering FOSI fragments (`atipl-eic.fos`) that are found in the `\lib` directory of your installation. XML document types are automatically recompiled.

To Build a Basic Line Numbering Application:

1. Build an ACL application that will be used to define the line numbering behavior you want to apply to the `atipl` tags in a document. You can provide specifications for each of the `atipl` tags. Detailed descriptions of the generic attributes for each tag are provided in the reference section of this chapter. The following list provides suggestions for your application:
   - If you want line numbers to restart at each new page, include a counter in your code that initializes at each atipl:startpage tag.
   - If you want line numbers to appear on every fifth line, include a counter in your code that sets the `attr1` on each atipl:startline tag that is divisible by `5`.
   - By default, line numbers are displayed in both the Edit view and composed print output. If you would like to limit line numbering to one media or the other, set the `atipl` variable to either `print` or `screen`. For example, to limit line numbers to composed print output, add the following line to your code:
     ```
     $atipl="print"
     ```
   - Generated text must be refreshed in order for the newly applied line numbers to be displayed in the Edit view. Add the following line to your code to automatically refresh generated text:

```
set gentext=off ; set gentext=on
```

2. Open an XML document and call the `layout::apply` function, passing your ACL application through as the first argument. The `layout::apply` function causes a series of composition and layout events to occur:
   a. A formatting pass is completed and a `.layout` file is generated, which specifies the structure of the document as it will appear in composed output, and defines where the `atipl` tags will appear. For more information about the layout file, please refer to The Layout file and document type on page 182.
   b. The `atipl` markup is added to your document.
   c. A second formatting pass is performed, your application is called and sets a series of common attributes on the `atipl` tags, which define the line numbers' appearance.
   d. The line numbers are displayed in your Edit view.

Line numbering ACL

Detailed information on the following ACL functions and set options can be found in the ACL documentation.

- `set pagelayoutmarkers` command
- `set protectpagelayout` command
- `oid_logical_mate` function
- `oid_find_valid_insert` function
- `layout::add` function
- `layout::clear` function
- `layout::apply` function
- `linenum` function

Line numbering application building reference

The following sections provide detailed information regarding the structure, conventions, and possible customization of the Arbortext line numbering framework.

Tag traversal and current tag conventions

Use the `pagelayoutmarkers` set option to control the display of the `atipl` markup, and the `protectpagelayout` set option to control whether or not it can be modified. The `caret` command will ignore `atipl` markup whenever it is not displayed, regardless of these command settings.

`oid` functions (for example, `oid_next` and `oid_prev`) do not recognize `atipl` markup whether or not it is displayed in the Edit window. Line numbering applications must be written to handle cases where `atipl` markup may interfere with tag or `oid` navigation.

The `atipl` singleton tags do not affect the balancing of selections, but they must be treated as pairs in other respects by all edit operations. This markup is ignored by the spell checking code, so that word fragments split by these tags are seen as a single word.

Deletion, either forward or backward, will ignore any `atipl` markup to the left of the cursor if it is not displayed. The deletion operation will fail if the markup is displayed and protected.

In the context of line numbering applications, the current tag is defined as the tag to the left of the cursor. The `atipl` tags can only be treated as the current tag when they are displayed.

The line numbering namespace

The line numbering namespace and associated markup (`atipl` tags) are described on the PTC Inc. namespace web site at:www.arbortext.com/namespace/index-of-arbortext-namespaces.html.

The `atipl` layout markup

The `atipl` tag set does not require a separate document type definition; it can be used with all document types. The definitions for these tags are in *`Arbortext-path`*`\lib\dtgen\atitag.cf`, and the default formatting is defined in FOSI fragment located at *`Arbortext-path`*`\lib\atipl-eic.fos`.

When the `layout::apply` function is called, a `.layout` file is created, using the structures defined in the `layout.dtd` to specify the composed layout of the document. The `atipl` singleton tags are then inserted as pairs around the document material that corresponds to the composed output structure they describe. Although `atipl` tags are singletons, if a particular tag cannot be inserted, its logical mate will not be inserted either. For example, if a `<atipl:startcolumn/>` tag cannot be inserted, the `<atipl:endcolumn/>` tag will also not be allowed.

Each start and end tag has a set of generic attributes. Every start tag also has a predefined set of attributes that correspond to the declared attributes of the matching element of the `layout.dtd`. For more detailed information on the `layout.dtd`, refer to section The Layout file and document type on page 182. The exceptions to this correlation are that the `oid` and `offset` attributes are not required, and the `<atipl:startfloat/>` tag has page, span, and column number attributes.

The `commonattr` entity in the `layout.dtd`

Each singleton pair described below is defined in the `commonattrs` entity which is declared in the `layout.dtd`.

`type`, `location`, `error` and generic attributes

```
<!ENTITY % commonattrs
 "type (forced|discretionary) "discretionary"
 location (inline|display) "inline"
 xmlns:atipl CDATA #IMPLIED
 error CDATA #IMPLIED
 attr1 CDATA #IMPLIED
 attr2 CDATA #IMPLIED
 attr3 CDATA #IMPLIED
 attr4 CDATA #IMPLIED
 attr5 CDATA #IMPLIED
 attr6 CDATA #IMPLIED
 attr7 CDATA #IMPLIED
 attr8 CDATA #IMPLIED
 attr9 CDATA #IMPLIED" >
```

The `type`, `location` and `error` attributes are used to control the method for generating formatting characteristics for an element and are set during the generation of layout markup. These attributes should not be modified.

The attributes `attr1` through `attr9` are generic attributes that can be used by the application writer to customize page layout applications. By convention, `attr1` is used to display automatically generated text, such as line numbers.

`startpage` and `endpage`

```
<!ELEMENT atipl:startpage EMPTY>
<!ATTLIST atipl:startpage
 number NMTOKEN #IMPLIED
 %commonattrs; >
<!ELEMENT atipl:endpage EMPTY>
<!ATTLIST atipl:endpage
 %commonattrs; >
```

The `startpage` markup indicates the start of a page, as determined by Arbortext Editor's formatting engine. The `number` attribute gives the sequential page number.

A folio may be set for the `attr1` attribute. It will appear as part of the line number in the format: `folio,\-\,lineno`.

The type of page break to force is controlled by the `attr2` attribute. Valid values are `next`, `verso`, and `recto`. The default is to not force a page break.

The `endpage` markup specifies the end of a page. If the `attr2` attribute is set to the `fill`, then underfull errors are not reported for this page and the page is not stretched if it is short.

`startspan` and `endspan`

```
<!ELEMENT atipl:startspan EMPTY>
<!ATTLIST atipl:startspan
 number NMTOKEN #IMPLIED
 columns NMTOKEN #IMPLIED
 %commonattrs; >
<!ELEMENT atipl:endspan EMPTY>
<!ATTLIST atipl:endspan
 %commonattrs; >
```

The start and end of a spanned column are specified by the `startspan` and `endspan` markup. For example, a page that contains two columns of text followed by a page wide table will consist of two spans. The span number, which is reset on every page, is indicated by the attribute `number`. The number of columns is indicated by `columns`.

`startcolumn` and `endcolumn`

```
<!ELEMENT atipl:startcolumn EMPTY>
<!ATTLIST atipl:startcolumn
 number NMTOKEN #IMPLIED
 %commonattrs; >
<!ELEMENT atipl:endcolumn EMPTY>
<!ATTLIST atipl:endcolumn
 %commonattrs; >
```

Columns within a span are indicated by the `startcolumn` and `endcolumn` markup. The `number` attribute indicates the column number. To force a column break, set `attr2` to `force`.

`startfloat` and `endfloat`

```
<!ELEMENT atipl:startfloat EMPTY>
<!ATTLIST atipl:startfloat
 class CDATA #IMPLIED
```

```
 flid CDATA #IMPLIED
 pagetype CDATA #IMPLIED
 %commonattrs; >

<!ELEMENT atipl:endfloat EMPTY>
<!ATTLIST atipl:endfloat
 %commonattrs; >
```

Floats are parts of a document that do not appear in a set order. Rather, floats appear at the top or bottom of a page, span, or column. The `class`, `flid`, and `pagetype` attributes refer to FOSI concepts associated with every float.

`startrow`, `endrow`, `startentry`, and `endentry`

```
<!ELEMENT atipl:startrow EMPTY>
<!ATTLIST atipl:startrow
 number NMTOKEN #IMPLIED
 %commonattrs; >
<!ELEMENT atipl:endrow EMPTY>
<!ATTLIST atipl:endrow
 %commonattrs; >
<!ELEMENT atipl:startentry EMPTY>
<!ATTLIST atipl:startentry
 number NMTOKEN #IMPLIED
 vspan NMTOKEN #IMPLIED
 hspan NMTOKEN #IMPLIED
 %commonattrs; >
<!ELEMENT atipl:endentry EMPTY>
<!ATTLIST atipl:endentry
 %commonattrs; >
```

The `startrow`, `endrow`, `startentry`, and `endentry` markup specifies the rows and columns of a table. The `number` attribute of a row is reset on every page, likewise the `number` attribute of an entry is reset in every row. The `vspan` and `hspan` attributes indicate that an entry is spanning. The former indicates the number of cells spanned vertically, the latter indicates the number spanned horizontally.

`startline` and `endline`

```
<!ELEMENT atipl:startline EMPTY>
<!ATTLIST atipl:startline
 typemask CDATA "1"
 %commonattrs; >
<!ELEMENT atipl:endline EMPTY>
<!ATTLIST atipl:endline
 hyphen NMTOKEN #IMPLIED
 %commonattrs; >
```

The `startline` and `endline` markup indicates the line breaks as defined by the formatting engine. The type of content in a line is indicated by the `typemask` attribute. The bits that may appear in a typemask indicate whether that content is plain or generated text, and are displayed in the following table:

Plain	Gentext	Content
`0x1`	`0x2`	characters
`0x4`	`0x8`	ruling
`0x10`	`0x20`	kern, kernto, hyphpt, hardsp, passthru
`0x40`	`0x80`	character fill (leader dots)
`0x100`	`0x200`	graphic
`0x400`	`0x800`	display equation
`0x1000`	`0x2000`	inline equation
`0x4000`	`0x8000`	forced line break

If a line ends with a hyphen, the character code of the hyphen is added to the `hyphen` attribute on the end tag.

The margin where the line numbers appear in the printed output is defined by the value of `attr2`. Legal values are `left` or `right`. The default is `right`.

The quadding of the number, relative to the page center, is defined by the value of `attr3`. This value may be `in` or `out`. The default value is `out`.

The end of a line, where a break is no longer discretionary, may require special treatment. Set `attr2` to `fill` on the end tag to end a line with a filler space that prevents an underfull error.

The Layout file and document type

The **Layout** document type defines the `.layout` file, which is produced by the Arbortext formatting engine and written to the `.aptcache` folder when line numbering is applied to a document. The `.layout` file specifies the structure of the document as it will appear in composed output, and defines where the `atipl` tags will appear.

The format of the `.layout` file is defined by the following document type definition. A typical declaration would be structured in this way:

```
<?xml version=1.0?>
<!DOCTYPE layout PUBLIC "-//Arbortext//DTD Layout 1.0//EN"
"layout/layout.dtd">
```

The common entities

The following entities are declared in the **Layout** DTD, and are used for declaring attributes that point back into the document or store dimensions.

```
<!ENTITY % oid "CDATA" > <!--vdid,df,genno-->
<!ENTITY % offset "NMTOKEN" > <!--zero based offset-->
<!ENTITY % dimen "CDATA" > <!--dimension in pt, e.g 1.25-->
```

Layout structure

A `.layout` file describes the page structures that result from the composition process applied to a source document. A typical `.layout` file will describe one or more `Page` structures.

The `Layout` element's `date` attribute holds the creation date in the form `DD-MM-YYYY`. The `file` attribute holds the system path of the source document, if available.

```
<!ELEMENT Layout (Page*)>
<!ATTLIST Layout
 date CDATA #IMPLIED
 file CDATA #IMPLIED >
```

Page level structures

A `Page` is a vertical layout container that holds an optional header, zero or more spans, and an optional footer. `Page-top` floats may appear after the header and `Page-bottom` floats may appear before the footer. Pages are numbered starting with `1` for the first page. The optional `oid` attribute indicates the element that forces the start of the page, if any.

`Header` and `Footer` are generated by the stylesheet. They may also contain information that is derived from the document or from the part of the document that is currently displayed. The header and footer are usually ignored by applications that move layout information back to the document.

`Span` is a horizontal layout container that holds one or more columns. For example, a page may have a title that spans the page, a three column span for text, and another one column span for a table. The optional `oid` attribute specifies the element in the document that forces the start of any such span.

Spans are numbered, starting with `1` for the first span on a page. The `columns` attribute specifies the maximum number of columns that a span can contain. Some of the columns in a span may be missing. The `width` attribute specifies the width of each column in a span measured in points.

`Column` is a vertical layout container that holds lines of galley material or tables. Columns are numbered, starting with `1` for the first column in a span. The `oid` attribute indicates the element that forces the start of any such column.

```
<!ELEMENT Page ((Header? , Float*, (Span+, Float*)?, Footer?))>
<!ATTLIST Page
 oid %oid; #IMPLIED
 number NMTOKEN #IMPLIED >
<!ELEMENT Header ((Line | Row)*)>
<!ELEMENT Footer ((Line | Row)*)>
```

```
<!ELEMENT Span (Float*, (Column+ , Float*)?)>
<!ATTLIST Span
 oid %oid; #IMPLIED
 number NMTOKEN #IMPLIED
 columns CDATA #IMPLIED
 width %dimen; #IMPLIED >

<!ELEMENT Column (Float*, ((Line | Row)+, Float*)?)>
<!ATTLIST Column
 oid %oid; #IMPLIED
 number CDATA #IMPLIED >
```

Floating structures

A `float` is a vertical container. It holds galley material that does not appear in sequence with the galley but rather in one of the many float areas available in the page layout. These areas are the top or bottom of the page, the top or bottom of any span, and the top or bottom of any column.

Floating material belongs to one of many float classes, and within a class multiple floats retain their galley order. For example, footnotes are floats that belong to the footnote class, and they appear in the page layout in the same order as they originally appeared in the instance.

The `oid` attribute indicates the element that starts the float.

The `class` attribute indicates the float class. The `class` also contains a float occurrence modifier. Repeating floats may appear many times, while `once` floats may only appear once. Applications may be written to ignore repeating floats and process `once` floats according to the `class` name.

The `flid` attribute (float identifier) provides a unique number for each float in a class.

The `pagetype` attribute defines the relationship between a float and its point of reference.

The `width` attribute specifies the width of the content.

```
<!ELEMENT Float ((Row | Line)*)>
<!ATTLIST Float
 oid %oid; #REQUIRED
 class CDATA #IMPLIED
 flid CDATA #IMPLIED
 pagetype CDATA #IMPLIED
 width %dimen; #IMPLIED >
```

Galley structures

Galley refers to the running text and tables that are laid out into columns during page composition.

`Row` is a horizontal container associated with tables that hold one or more entries. A table is made up of rows, some of which are header rows and some of which are footer rows. The `oid` attribute indicates the element that starts the row.

`Entry` is a vertical container that holds the material that appears in a table cell. This material is typeset using the width of the entry (given by the `width` attribute). An entry may span columns (`hSpan`) and rows (`vSpan`). The `oid` attribute indicates the element that starts the entry.

`Line` is a horizontal container that holds text, graphics, or equations. Line numbering applications focus on the start and end of each line. If an element forced the start of a line, this is indicated by the `oid` attribute.

```
<!ELEMENT Row (Entry+)>
<!ATTLIST Row
 oid %oid; #IMPLIED
 number NMTOKEN #IMPLIED >
<!ELEMENT Entry ((Line | Row)*)>
<!ATTLIST Entry
 oid %oid; #IMPLIED
 number NMTOKEN #IMPLIED
 hSpan NMTOKEN #IMPLIED
 vSpan NMTOKEN #IMPLIED
 width CDATA #IMPLIED >
<!ELEMENT Line ((Text | Graphic | Equation)*)>
<!ATTLIST Line
 oid %oid; #IMPLIED
 y %dimen; #IMPLIED >
```

Text level structures

Text level structures are the visible objects that appear on the page. They include text, graphics, and equations. Rules and leaders are ignored by line numbering applications.

`Text` refers to a sequence of characters that are displayed one font. The concept of a word does not exist, because a string of characters includes space characters. If implemented, the `text` element may contain a string of characters as PCDATA, otherwise it is empty.

The `oid`, `sOffset`, and `eOffset` parameters can be used to locate the exact substring in the source document that corresponds to a `text` element. If the text fragment ended in a discretionary hyphen (inserted by the formatting engine), the hyphen character is indicated by the `hyphen` attribute.

`Graphic` is an object that will be rendered as an image based on data outside of the document instance (for example, a `.gif` file). The `file` attribute gives the location of the file.

`Equation` is an object that will be rendered as a mathematical equation by the Arbortext formatting engine. Equations may be of two types, either display or inline.

```
<!ELEMENT Text (#PCDATA) >
<!ATTLIST Text
 oid %oid; #REQUIRED
 sOffset %offset; #IMPLIED
 eOffset %offset; #IMPLIED
 hyphen NMTOKEN #IMPLIED
 x %dimen; #IMPLIED >
<!ELEMENT Graphic EMPTY>
<!ATTLIST Graphic
 oid %oid; #REQUIRED
 x %dimen; #IMPLIED
 file CDATA #IMPLIED >
<!ELEMENT Equation EMPTY>
<!ATTLIST Equation
 oid %oid; #REQUIRED
 x %dimen; #IMPLIED
 type (display|inline) #IMPLIED >
```

IV

Interfaces

16

Interface Overview

The AOM supports most of the DOM interfaces developed by the W3C, several Arbortext extensions to the DOM interfaces, and many additional Arbortext interfaces for features that are not part of the DOM. Refer to Introduction to the Document Object Model (DOM) on page 36 for a list of supported DOM specifications.

The interface descriptions use the DOM conventions in presenting a language-neutral definition of the list of constants (enumerations), attributes (properties), and methods implemented for each interface. For some language bindings, the enumeration (constant) names are available as global `typedefs` (for example, COM C++), as `static final` constants (Java, JavaScript), or only available as numeric values (JScript and VBScript, currently). Attributes (or properties) in some language bindings are translated to `set`*`Xxx`* and `set`*`Xxx`* methods. For example, the `Application.activeDocument` attribute is obtained by calling the `Application.getActiveDocument()` method in Java. Read-only attributes, as noted in the `Access` table entry of each attribute description, only have a `get`*`Xxx`* method in these language bindings. (Refer to the Index terms “attributes”, “enumerations”, and “methods” for alphabetical listings of each, respectively.)

The descriptions of the W3C interfaces in the following chapters are taken from their respective W3C specifications. Each description provides a reference to its W3C specification.

In the W3C interface descriptions, the `DOMString` type is a string of 16-bit Unicode characters, the same as the `String` type in the other interface descriptions. Throughout the documentation consider references to HTML or XML to also include SGML.

Square braces (**[ ]**) denote optional trailing parameters which may be omitted in most script bindings. Also, the AOM provides method overloads in the Java binding so that optional parameters may be omitted.

The AOM supports the following interfaces:

Interface	Description
AbstractView	(W3C) A base interface that all views shall derive from.
Acl	Represents the ACL (Arbortext Command Language) interpreter, allowing the AOM programmer to request that a string be executed as an ACL command or evaluated as an ACL function.
ActivexEvent	Provides specific contextual information associated with `Activex` events.
ADocument	The Arbortext extension to the W3C DOM **Document** interface.
ADocumentType	Arbortext extensions to the W3C DOM **DocumentType** interface
AEditEvent	Provides specific contextual information associated with the **EditEvent** extension.
AElement	The Arbortext extension to the W3C DOM **Element** interface.
AEvent	The Arbortext extension to the W3C DOM **Event** interface.
ANode	The Arbortext extension to the W3C DOM **Node** interface.
Application	Provides access to Arbortext Editor and Arbortext Publishing Engine global functionality. (That is, features that are not associated with any document, document type, or document component.) There is only one `Application` object instantiation in existence.
ARange	The Arbortext extension to the W3C DOM **Range** interface.
Attr	(W3C) An attribute in an **Element** object.
CDATASection	(W3C) Used to escape blocks of text containing characters that would otherwise be regarded as markup.

Interface	Description
CharacterData	(W3C) Extends **Node** with a set of attributes and methods for accessing character data in the DOM.
Comment	(W3C) Inherits from **CharacterData** and represents the content of a comment, for example, all the characters between the starting `<!--` and ending `-->`.
Component	The base interface for all window components.
Composer	Represents a composition pipeline defined by a Composer Configuration File (CCF).
ControlEvent	Provides specific contextual information associated with `Control` events.
Dialog	Extends the **Window** interface.
Document	(W3C) Represents the entire HTML or XML document.
DocumentEvent	(W3C) Provides a mechanism by which the user can create an **Event** of a type supported by the implementation.
DocumentFragment	(W3C)A "lightweight" or "minimal" **Document** object.
DocumentRange	(W3C) Provides a mechanism to create **Range** objects for a document.
DocumentType	(W3C) Each **Document** has a doctype attribute whose value is either null or a **DocumentType** object.
DocumentView	(W3C) Implemented by **Document** objects in DOM implementations supporting DOM **View**s.
DOMImplementation	(W3C) Provides a number of methods for performing operations that are independent of any particular instance of the document object model.
Element	(W3C) The Element interface represents an element in an HTML or XML document.

Interface	Description
Entity	(W3C) This interface represents an entity, either parsed or unparsed, in an XML document.
EntityReference	(W3C) **EntityReference** objects may be inserted into the structure model when an entity reference is in the source document, or when the user wishes to insert an entity reference.
Event	(W3C) Used to provide contextual information about an event to the handler processing the event.
EventListener	(W3C) The primary method for handling events.
EventTarget	(W3C) Implemented by all **Node**s in an implementation which supports the DOM Event Model. Also implemented by all **Component**s in the AOM implementation.
MenuBar	Represents a menu bar.
MenuEvent	Provides specific contextual information associated with Menu events.
MenuItem	Represents a menu item.
MouseEvent	(W3C) Provides specific contextual information associated with Mouse events.
MutationEvent	(W3C) Provides specific contextual information associated with Mutation events.
NamedNodeMap	(W3C) Objects implementing the **NamedNodeMap** interface are used to represent collections of nodes that can be accessed by name.
Node	(W3C) The primary datatype for the entire Document Object Model.
NodeList	(W3C) Provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented.

Interface	Description
Notation	(W3C) Represents a notation declared in the DTD.
ProcessingInstruction	(W3C) Represents a processing instruction. Used in XML as a way to keep processor-specific information in the text of the document.
PropertyMap	Provides the abstraction of a collection of typed objects associated with string keys.
Range	(W3C) Represents a range of content in a **Document**, **DocumentFragment**, or **Attr**.
ScriptContext	Provides methods to load and run scripts using the Microsoft Windows Scripting engine in separate contexts. This interface is only available in the COM binding of the AOM.
StringList	Provides the abstraction of an ordered collection of **String**s, without defining or constraining how this collection is implemented.
TableCell	Represents a single cell in a table.
TableColumn	Represents a column of cells.
TableGrid	Represents a table grid which is a rectangular array of cells.
TableMulticell	Represents a rectangular array of spanned cells in a table.
TableObject	Base class for all table objects.
TableObjectStore	A **TableObjectStore** contains a collection of **TableObject**s all from the same document.
TableRectangle	Represents a rectangle of cells.
TableRow	Represents a row of cells.
TableRule	Represents a rule.
TableSet	A collection of one or more **TableGrid**s, each of which is a rectangular array of **TableCell**s.
TableTilePlex	Used to represent a table selection.

Interface	Description
Text	(W3C) Inherits from **CharacterData** and represents the textual content (termed character data in XML) of an **Element** or **Attr**.
ToolBarEvent	Provides specific contextual information associated with `ToolBar` events.
UIEvent	(W3C) Provides specific contextual information associated with User Interface events.
View	A subclass of **AbstractView**, representing a view of an associated **Document**.
Window	Represents a top level window frame which is created by Arbortext Editor.
WindowEvent	Provides specific contextual information associated with **Window** events.

The AOM supports the following Arbortext PE Application interfaces:

Interface	Description
CCComposer	Describes a single composer (`.ccf` file) installed on the Arbortext Publishing Engine server.
CCCompositionParameter	Describes a single parameter to a Arbortext Content Pipeline composer (`.ccf` file).
CCDoctype	Describes a single document type installed on a Arbortext Publishing Engine server.
CCDocumentComposer	Describes a composer associated with a document type installed on a Arbortext Publishing Engine server.
CCFrameset	Describes a frameset that is installed on a Arbortext Publishing Engine server.
CCPathEntry	Describes a single directory on a server path list.
CCStylesheet	Describes a stylesheet installed on the Arbortext Publishing Engine server.

Interface	Description
CompositionConfiguration	Provides information about a Arbortext Publishing Engine server's composition capabilities.
E3Application	Creates an object that runs in each Arbortext PE sub-process and is called by the Arbortext Publishing Engine to process HTTP requests.
E3ApplicationRequest	Provides request information for a Arbortext Publishing Engine Application.
E3ApplicationResponse	Provides an object to assist a Arbortext Publishing Engine Application in sending a response to the HTTP or SOAP client.
E3ClientCompositionExtension	Describes an object that provides composition type-specific pre- and post-processing routines for the Arbortext Publishing Engine Composition Client.
E3Config	Passes information to a Arbortext Publishing Engine Application during initialization.
E3ServerComposer	Describes an object that handles composition operations on a Arbortext Publishing Engine server. “Composition" includes transforming an input JAR file into an output JAR file.
E3ServerCompositionExtension	Extends the Arbortext Publishing Engine Server Composition Application.
E3ServerCompositionParameter	Describes a parameter passed to or returned by an **E3ServerCompositionRequest**.
E3ServerCompositionRequest	Describes the request for a composition operation to be performed by the Arbortext Publishing Engine server composition application.

Interface	Description
E3ServerCompositionResult	Describes the result of a composition operation under the Arbortext Publishing Engine server composition application.
E3Tracer	Creates entries in the Arbortext Publishing Engine Server Composition trace files.

17

W3C AbstractView interface

The `AbstractView` interface is defined in the W3C Document Object Model (DOM) Level 2 Views Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113.)

A base interface that all views shall derive from.

document attribute

The source `DocumentView` of which this is an `AbstractView`.

`document`	
Access	read-only
Returns	`DocumentView`

18

Acl interface

The `Acl` interface represents the ACL (Arbortext Command Language) interpreter. It allows the AOM programmer to request that a string be executed as an ACL command or evaluated as an ACL function. The `Acl` interface also provides methods for converting from ACL OIDs to DOM nodes and from ACL document identifiers to DOM Document nodes.

DOMDocument method

Returns the DOM Document object corresponding to an Arbortext document ID. The desired document must be open in Arbortext Editor or Arbortext Publishing Engine before calling this method, but the document does not need to be visible in a window. Developers can obtain the document identifier they need by using the `Eval` method to call an ACL function such as `current_doc`.

`DOMDocument`(docId)	
Parameters	`long` *docId* The ACL document identifier of a document. If zero (0), the method uses the returned value of the ACL function `current_doc`.
Returns	`Document`. The DOM document object.

DOMOID method

Returns the DOM Node associated with the supplied ACL object identifier `oid`.

This method is useful for creating a DOM node object from a portion of a document instead of the entire document. The desired document must be open in Arbortext Editor or Arbortext Publishing Engine before calling this method. The object identifier `oid` can be obtained by using the `Eval` method to call an ACL function such as `oid_caret`.

`DOMOID`(oid)	
Parameters	`String` *oid* The ACL object identifier.
Returns	`Node`. The DOM Node object. If `oid` is invalid, returns 0.

Eval method

Evaluates a string as an ACL expression and returns the result of the evaluation as a string. The string to evaluate must contain an expression. For example:

```
2+2
```

or

```
tbl_oid_cell(oid_caret(),oid_caret_pos())
```

Variable substitution in the expression string occurs on the ACL side of the AOM interface, not on the client side. You can include ACL variables in the expression string. However, do not include variables native to the client program.

`Eval`(expression)	
Parameters	`String` *expression* The ACL expression to evaluate.
Returns	`String`. The result of the evaluated expression as a string.

Execute method

Executes a string as an ACL command. The return value varies depending on the interface.

Variable substitution in the expression string occurs on the ACL side of the AOM interface and not on the client side. You can include ACL variables in the expression string. However, do not include variables native to the client program.

`Execute`(command)	
Parameters	`String` *command* The ACL command to execute.
Returns	`String`. The result depends on the interface

GetCMSObject method

Returns a `CMSObject` object equivalent to the given ACL dobj id.

`GetCMSObject`(objectId)	
Parameters	`long` *objectId* Represents a valid ACL object id.
Returns	`CMSObjectCMSObject`. object equivalent to the given ACL dobj id. `null` will be returned if the given ACL dobj id is invalid.

GetCMSSession method

Returns the `CMSSession` object associated with the given ACL session id.

This does **not** support the default (file-system) session id value of `0`.

`GetCMSSession`(sessionId)	
Parameters	`long` *sessionId* Represents an active ACL session id.
Returns	`CMSSession`. CMSSession object associated with the given ACL session id. `null` will be returned if the given ACL session id is invalid.

GetVar method

Returns the value of an ACL scalar variable as a string.

`GetVar`(name)	
Parameters	`String` *name* The name of the ACL variable to retrieve. If the variable is not qualified with a package name, the`main` package is used.
Returns	`String`. The value of the specified ACL variable as a string.

GetWindow method

Returns the AOM `Window` object corresponding to an Arbortext window ID. Developers can obtain the window identifier they need by using the `Eval` method to call an ACL function such as `current_window`.

`GetWindow`(winId)	
Parameters	`long` *winId* The ACL window identifier of a window. If zero (`0`), the method uses the returned value of the ACL function `current_window`.
Returns	`Window`. The AOM window object.

SetVar method

Sets the value of an ACL scalar variable to the specified string.

`SetVar`(name, value)	
Parameters	`String` *name* The name of the ACL variable to set. If the variable is not qualified with a package name, the`main` package is used. `String` *value* The new value for the ACL variable. There are no size limits (beyond available memory) on the length of the string.
Returns	`void`
Throws	AOMException Raised if the ACL variable is read-only.

19

ActivexEvent interface

The `ActivexEvent` interface provides specific contextual information associated with `Activex` events.

initActivexEvent method

Initializes the value of an `ActivexEvent` created through the `Window createEvent` method. This method should only be called before the `ActivexEvent` has been dispatched with the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initActivexEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

20

ADocument interface

The Arbortext extension to the W3C DOM `Document` interface.

ATISelectionType enumeration

The `selectionType` attribute describes the type of selection in the `Document` and has one of the following values.

The `ATISelectionType` enumeration has the following constants of type `short`.

NO_SELECTION = 0
There is no selection.

TEXT_SELECTION = 1
There is a text selection.

TABLE_SELECTION = 2
There is a table selection.

MarkupType enumeration

The `MarkupType` enumerated type defines the values for the `markupType` attribute and has the following constants:

The `MarkupType` enumeration has the following constants of type `int`.

NO_MARKUP = 0
The document does not use XML, SGML, or HTML markup. That is, it is untagged.

XML_MARKUP = 1
The document uses XML markup.

SGML_MARKUP = 2
The document uses SGML markup.

HTML_MARKUP = 3
The document is an HTML document. This value is not used for XHTML documents, which have `markupType` of `XML_MARKUP`.

SaveFlags enumeration

The `SaveFlags` enumerated type is used to construct the `flags` parameter to the `save` method, by ORing options from the following list:

The `SaveFlags` enumeration has the following constants of type `int`.

SAVE_CT_ORIG = 0x0001
For documents with change tracking markup, save as if all changes are rejected (original view).

SAVE_CT_LATEST = 0x0002
For documents with change tracking markup, save as if all changes are accepted (latest view).

SAVE_CT_ALL = 0x0004
For documents with change tracking markup, save as if all changes are pending (highlighted view).

If none of the SAVE_CT_xxx flags are set, the document is written as specified by the `Application.getOption("writechangetracking")` setting. If SAVE_CT_ORIG is specified with either of the other options, SAVE_CT_ORIG is obeyed. If SAVE_CT_LATEST and SAVE_CT_ALL are both specified, SAVE_CT_LATEST is obeyed.

SAVE_SGML = 0x0008
Write the document as an SGML document.

SAVE_UNTAGGED = 0x0010
Write a text-only version of the document.

SAVE_XML = 0x0020
Write the document as XML.

If one of the SAVE_SGML, SAVE_UNTAGGED, or SAVE_XML options is not specified, an SGML document is written as SGML and an XML document is written as XML. If more than one option is specified and SAVE_XML is specified, it is obeyed; otherwise, SAVE_SGML is used.

SAVE_NOHEADER = 0x0040
Removes the DOCTYPE header and internal subset including any private ENTITY declarations.

SAVE_NOPI = 0x0080
Removes Arbortext-specific processing instructions.

If not specified, behavior is controlled by the `Application.getOption("writepi")` setting.

SAVE_EOC = 0x0100
Enables entity output conversion.

SAVE_NOEOC = 0x0200
Suppresses entity output conversion.

If neither SAVE_EOC nor SAVE_NOEC is specified, entity output conversion is controlled by the `Application.getOption("entityoutputconvert")` setting. If both are specified, entity output conversion is enabled.

SAVE_NAC_ENTREF = 0x0400
Writes non-ASCII characters as character entity references.

SAVE_NAC_CHAR = 0x0800
Writes non-ASCII characters as characters in the target encoding.

SAVE_NAC_NUMREF = 0x1000
Writes non-ASCII characters as numeric character references.

If none of the SAVE_NAC_xxx options are specified, behavior is controlled by the `Application.getOption("writenonasciichar")` setting. If more than one is specified, SAVE_NAC_ENTREF takes precedence if specified; otherwise SAVE_NAC_CHAR takes precedence if specified.

SAVE_NOBREAK = 0x2000
Used internally for HTML output.

SAVE_FLATTEN_FILE = 0x4000
Expands all file entities recursively.

SAVE_FLATTEN_TEXT = 0x8000
Expands all text entities recursively.

SAVE_NON_FRAGMENT = 0x10000
Writes a non-fragment header if possible.

SAVE_FLATTEN_INCLUDE = 0x20000
Expands all XInclude references recursively.

CloneFlags enumeration

The following bit constants are used with the `flags` argument of the `cloneDocument()` method.

The `CloneFlags` enumeration has the following constants of type `int`.

CLONE_NO_CONTENT = 0x01
No content will be cloned. This will result in an empty document.

CLONE_RESOLVE_CT = 0x02
Resolve any change tracking markup according to the value of the `viewchangetracking` option for the current view of the source document. If there is no view setting associated with the source document, the global value of the `viewchangetracking` option will be used.

The `viewchangetracking` option interacts with this function in the following way:

`original` — The cloned document will have the original markup (changes not applied) but no change tracking markup.

`changesapplied` — The cloned document will have the latest markup (changes applied) but no change tracking markup.

`changeshighlighted` — The cloned document will be as if `CLONE_RESOLVE_CT` were not set. It will have the change tracking markup (no data is lost; changes are still tracked).

CLONE_NO_ENT_DECLS = 0x04
Makes the empty document not inherit entity declarations from the source document. Only obeyed if `CLONE_NO_CONTENT` is also specified.

CLONE_XML = 0x08
Force clone to be XML even if the source is SGML. Only obeyed if source document is made up of markup (not pure text).

CLONE_CARET = 0x10
Include the source document's caret position in the cloned content. Only obeyed if `CLONE_NO_CONTENT` is **not** specified.

CLONE_LOCATION = 0x20
Include every block oid in the source document as a pi in the cloned content. Only obeyed if `CLONE_NO_CONTENT` is **not** specified. The PI has the format of `<?APTCOMP EPIC _OID_?>` where `_OLD_ = (dfid, generate_no, docid)`

CLONE_NAME = 0x40
Sets the name of the cloned documnt to the name of the source document.

ModifyRefFlags enumeration

The `ModifyRefFlags` enumerated type is used to construct the `flags` parameter to the `modifyReferences` method by ORing any of the following options.

The `ModifyRefFlags` enumeration has the following constants of type `int`.

MODIFYREF_NO_CUSTOMREF = 0x0001
Indicates that the burst configuration file associated with the doctype of the document given to the `modifyReferences` method should not be consulted in order to determine which DOM nodes are considered customref references. The result of this flag is that no customref references will be modified.

MODIFYREF_NO_GRAPHICS = 0x0002
Indicates that neither the Arbortext Styler stylesheet nor DCF file associated with the document or the doctype of the document given to the `modifyReferences` method should be consulted in order to determine

which DOM nodes are considered “graphics”. The result of this flag is that no graphics references will be modified.

MODIFYREF_NO_FILEENTS = 0x0004
Indicates that file entity references will not be modified.

MODIFYREF_NO_XINCLUDES = 0x0008
Indicates that XInclude references will not be modified.

CMSObjects attribute

Returns an collection of all the objects in this document. The objects in this collection may be in any order but each will be present exactly once. Note that if a document contains a given child object in two locations then the returned collection will contain two objects; one for each reference. Each object will reference the same repository object but, for example, will have different `start` and `end` values associated with them.

`CMSObjects`	
Access	read-only
Returns	`CMSObjectList`
Get throws	CMSException Raised for any error.

aclId attribute

An integer constant uniquely identifying the document. This is the value that would be returned by the ACL function `current_doc` if the document were current.

`aclId`	
Access	read-only
Returns	`long`

directory attribute

The directory associated with the document. For documents read from the file system, this is the directory part of the `documentURI` attribute, excluding the name, and expressed as a file system path not as a URI. If the document has no directory, for example, a new document created from a template and not yet saved, this is the null string. A document created by calling `cloneNode` on another `Document` node inherits this attribute.

This attribute is read-only. However, changing the `documentURI` attribute will also change the value of the `directory` attribute.

directory	
Access	read-only
Returns	String

insertionPoint attribute

A collapsed DOM Range indicating the current location of the cursor.

insertionPoint	
Access	read-write
Returns	Range
Set throws	DOMException INVALID_STATE_ERR: Raised if the Range has already been detached.

markupType attribute

An integer constant indicating the type of markup used by the document. One of the following values: XML_MARKUP, SGML_MARKUP, HTML_MARKUP, or NO_MARKUP.

markupType	
Access	read-only
Returns	MarkupType

modified attribute

A boolean indicating whether the document has been modified. This attribute is reset when the document in saved.

modified	
Access	read-write
Returns	boolean

name attribute

The name of the document or a null string if the document was created without a name. For documents read from the file system, the name is the base name of the documentURI, including the extension, if any.

name	
Access	read-write
Returns	String

optionNames attribute

A `StringList` containing the names of all document-scope Arbortext `set` options.

`optionNames`	
Access	read-only
Returns	`StringList`

properties attribute

A `PropertyMap` object containing user-defined properties for the document. The properties are stored at the beginning of the XML file as processing instructions.

`properties`	
Access	read-only
Returns	`PropertyMap`

selectionType attribute

An integer constant indicating whether there is no selection (`NO_SELECTION`), a text selection (`TEXT_SELECTION`), or a table selection (`TABLE_SELECTION`).

`selectionType`	
Access	read-only
Returns	`ATISelectionType`

tables attribute

A `TableObjectStore` containing all of the `TableSet`s in the document. If there are no tables in the document, an empty store is returned.

`tables`	
Access	read-only
Returns	`TableObjectStore`

tableSelection attribute

A `TableTilePlex` representing the current table selection. If there is no table selection, the value of `tableSelection` is an empty `TableTilePlex`.

tableSelection	
Access	read-write
Returns	TableTilePlex

textSelection attribute

A DOM Range indicating the current text selection. If there is no text selection, the value will be the same as insertionPoint.

If the text selection is set to a collapsed range, the selection is cleared. The insertion point is not changed.

textSelection	
Access	read-write
Returns	Range
Set throws	DOMException INVALID_STATE_ERR: Raised if the Range has already been detached.

canRenameNode method

Tests whether an existing node of type ELEMENT_NODE or ATTRIBUTE_NODE can be renamed such that the resulting node is compliant with VAL_SCHEMA validity type.

canRenameNode(node, namespaceURI, qualifiedName)	
Parameters	Node *node* The Node to be renamed. String *namespaceURI* The new namespace URI. String *qualifiedName* The new qualified name.
Returns	unsigned short. A validation state constant.

cloneDocument method

Creates a completely independent copy of this document. The cloned document will have no Document.documentURI or ADocument.name attributes set for it. However, the ADocument.directory attribute will be identical to the source document so that relatively-referenced resources (such as graphic files) will be correctly resolved in the context of the cloned document.

Note

You should avoid using the `Document.documentURI` attribute to give the cloned document a URI identical to the source document because any subsequent changes made to either document will be reflected in the other document.

`cloneDocument( [flags] )`	
Parameters	`int` *flags* [optional] A bitmask constructed by ORing some combination of the following constants: `CLONE_EMPTY`, `CLONE_RESOLVE_CHANGE_TRACKING`, `CLONE_NO_ENT_DECLS`, `CLONE_XML`, `CLONE_CARET`. See the descriptions of these constants for more information.
Returns	`Document`. Cloned document.
Throws	CMSException Raised for any error.

close method

Closes the document, freeing all associated memory and system resources (for example, file handles). This method actually decrements the reference count for the document and does not free resources until the reference count becomes zero. The reference count is incremented when the document is associated with a `View` object.

`close()`	
Parameters	None
Returns	`boolean`. Returns `true` if the document was actually closed. `false` otherwise. Since an exception is thrown on an error, this will always be `true` when no exception is thrown.
Throws	AOMException Raised if the method detects any error.

editBegin method

The `editBegin` and `editEnd` methods provide a mechanism to bracket a series of document changes which may optionally be rolled back. Before beginning a series of changes, call `editBegin` for this document. At the end of the changes, call `editEnd` to either commit the changes or to roll them back by specifying `false` as the commit parameter.

Multiple calls may be made to `editBegin` before an `editEnd` call, for example if one top-level script calls another as part of its implementation. In this case, the changes are not committed or rolled back until the outermost `editEnd` call is made. All changes since the first `editBegin` call will be rolled back if any nested call to `editEnd` or the outermost `editEnd` call specifies `false` as the commit parameter.

For example, in JavaScript:

```
doc.undoBoundary("Big Changes");
doc.editBegin();
var commit = true;
try {
 doBigChanges();
} catch (e) {
 commit = false;
}
doc.editEnd(commit);
```

This example assumes `doBigChanges` or a method it calls throws an exception if it detects an error condition after making some document changes which should then be discarded.

Note

Each call to `editBegin` must be matched with a call to `editEnd`. Failure to do so may cause unexpected behavior until Arbortext Editor or Arbortext Publishing Engine is restarted. For language bindings that support exceptions, DOM or AOM calls between `editBegin` and `editEnd` calls must be wrapped in a try/catch block so that `editEnd` is called if an exception is raised.

`editBegin( )`	
Parameters	None
Returns	`void`

editEnd method

This method commits or rolls back the changes made to the document since the matching `editBegin` call. The commit or roll back does not actually happen until the outermost `editEnd` call is made. Refer to the description of `editBegin` for details.

Note

Each call to `editBegin` must be matched with a call to `editEnd`. Failure to do so may cause unexpected behavior until Arbortext Editor or Arbortext Publishing Engine is restarted. For language bindings that support exceptions, DOM or AOM calls between `editBegin` and `editEnd` calls must be wrapped in a try/catch block so that `editEnd` is called if an exception is raised.

`editEnd`(commit)	
Parameters	`boolean` *commit* If `true`, specifies that the change should be committed. If `false`, changes will be rolled back (undone).
Returns	`void`

generateEntityName method

Generates an entity name suitable for use with this document. If no logicalId is given (or if it doesn't map to an active session), a random number is used to create an entity name which is currently not in use by this document. Otherwise the associated CMS adapter session will be called to produce the entity name. The adapter guarantees that the returned entity name will be unique as per the given logicalId. Thus, if given the same logicalId twice, this may return the same entity name twice. However, if given different logicalId's, this will return different entity names.

`generateEntityName`([logicalId])	
Parameters	`String` *logicalId* [optional] Logical ID used to ask an associated CMS adapter session to generate the unique name.
Returns	`String`. Unique entity name suitable for use with this document.
Throws	CMSException Raised for any error.

getElementsByAttribute method

Returns a `NodeList` of all descendant `Elements` that match the given attribute name and attribute value, in the order in which they are encountered in a pre-order traversal of this `Document` tree.

`getElementsByAttribute`(name, value, selector)	
Parameters	`String` *name* Specifies the name of the attribute to match. The value `"*"` matches all attribute names. `String` *value* Specifies the value of the attribute to match. `ATIElementAttributeSelector` *selector* Specifies how the attribute value should be matched. When selector is `0`, the value parameter is ignored. When selector is `1`, only the elements that match both the name and the value are included.
Returns	`NodeList`. A list of matching element nodes.
Throws	DOMException SYNTAX_ERR: If selector is invalid. INVALID_CHARACTER_ERR: If name is namespace qualified.

getElementsByAttributeNS method

Returns a `NodeList` of all descendant `Elements` that match the given attribute namespace URI, local name, and attribute value, in the order in which they are encountered in a pre-order traversal of this `Document` tree.

`getElementsByAttributeNS`(namespaceURI, localName, value selector)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to retrieve. The value `"*"` matches all namespaces. `String` *localName* Specifies the local name of the attribute to match. The value `"*"` matches all local attribute names. `String` *value* Specifies the value of the attribute to match. `ATIElementAttributeSelector` *selector* Specifies how the attribute value should be matched. When selector is `0`, the value parameter is ignored. When selector is `1`, only the elements that match both the name and the value are included.

Returns	`NodeList`. A list of matching element nodes.
Throws	DOMException SYNTAX_ERR: If selector is invalid. INVALID_CHARACTER_ERR: If localname is namespace qualified.

getOption method

This method returns the value of the Arbortext `set` option, scoped to this document.

`getOption`(name)	
Parameters	`String` *name* Specifies the option name, which must be a document-scope option.
Returns	`String`. The string value of the option, or null if `name` is not a valid option name. Boolean values return `on` or `off`.

modifyReferences method

This method will replace references within the given `ADocument`. The references to be replaced are those listed as keys in the given `PropertyMap`, and will be replaced by the value of each associated `PropertyMap` key. If the given `ADocument` contains any inclusions (such as file entities or XIncludes), unlike `IOHost::modifyReferences`, this method will descend into those inclusions in order to update any references that might be found in their content if the reference is found as a key in the given `PropertyMap`.

What is considered an “inclusion”, as far as this method is concerned, is limited to file entities and XIncludes. Any elements or attributes of elements which are encountered that match a customref burst configuration file rule (as found in the burst configuration file associated with the doctype of the `Document` or `CMSObject` to which the scrutinized node belongs) is not considered an “inclusion” by this method since customref is a referencing mechanism and not intended for inline inclusions. Any matching customref references will be replaced by this method, but since customref is not considered an “inclusion ” mechanism, this method will not open the file or logical id the customref references in order to descend into its contents.

All keys in the `PropertyMap` that reference the file system will be made a canonicalized universal name before any lookups occur. Also, each reference that is to be looked up in the `PropertyMap` that is a filesystem reference will also be temporarily made into a canonicalized universal name before the lookup occurs. By making all filesystem references canonicalized universal names, the caller will

be assured that multiple references that use different conventions but still reference the same filesystem location are actually recognized as the same reference. No such manipulation will be made to CMS logical ID references.

If the `MODIFYREF_NO_CUSTOMREF` flag is not included in the flags parameter, any elements or attributes of elements that are encountered that match a customref burst configuration file rule (as found in the burst configuration file associated with the doctype of the `Document` or `CMSObject` to which the scrutinized node belongs) will be recognized as a reference and as such will be modified as long as that reference is listed as a key in the given `PropertyMap`. If the mode of the customref rule is “dita-full”, then the reference will be replaced with the value of the relevant `PropertyMap` key, appended with any DITA fragment identifier (including the leading “#”) copied from the original reference. All customref rules whose mode is “dita-partial” are always ignored and never replaced by this method, even if the reference of the “dita-partial” customref is found as a key in the given `PropertyMap`.

`Documents` and `CMSObjects` have a notion of whether or not they contain unsaved modifications. The modified state of the `Document` or `CMSObject` to which the given `DocumentFragment` belongs will be preserved by this method.

`modifyReferences`(map, flags)	
Parameters	`PropertyMap` *map* The given `PropertyMap` that associates the list of references to be replaced with the references to replace them with. Any values in the `PropertyMap` that are numbers (TYPE_NUMBER) or `StringLists` (TYPE_STRINGLIST) will be ignored. `int` *flags* Specifies which constraints are placed upon the `modifyReferences` processing. The value is determined through a bit-wise OR of the `ModifyRefFlags` constants.
Returns	`int`
Throws	AOMException Raised if an error occurs. If during processing, a reference named in the `PropertyMap` cannot be updated for whatever reason, the processing will stop immediately and an exception will be thrown.

redo method

The redo method reverses the change made by the last undo. A series of consecutive undos may be reversed by the corresponding number of redos. Redo operations do not get added to the undo history.

`redo`()	
Parameters	None
Returns	`void`
Throws	AOMException Raised if the method detects an error, for example, when the last change was not an undo or redo.

save method

Saves this document.

`save`([flags [, path [, encoding [, publicId [, systemId]]]]])	
Parameters	`int` *flags* [optional] A bitmask that specifies save options. Constructed by ORing the bits from the `SaveFlags` enumeration. `String` *path* [optional] Specifies the path name of the output file. It may be any of the following values: • The name of a file. If it exists, it is silently rewritten. • A WebDAV URL (Windows only). • (UNIX only) A dash (`"-"`) indicating standard input. • An asterisk (`"*"`) indicating the message window. • (UNIX only) a UNIX pipeline (`"|"`). If the `path` is omitted or a null string, the document is saved to the original path name or Logical ID. If the document does not have a path, or if the path is not writable (for example, the document was read from an http: URL not on a WebDAV server, or the backing object was not checked out from the DMS), the method raises an exception. `String` *encoding* [optional] Determines the encoding of the file being written. This parameter overrides the encoding declaration in the document. The following table lists the valid strings for the encoding parameter.

	Adobe-Standard-Encoding ISO-8859-9 ISO-8859-1 EUC-JP ISO-8859-1-Windows-3.1-Latin-1* Shift_JIS ISO-8859-2 Big5 ISO-8859-3 GB_2312-80 ISO-8859-4 KSC_5601 ISO-8859-5 UTF-8 ISO-8859-7 US-ASCII ISO-8859-8 ISO-10646-UCS-2 *Windows only If encoding is `null` or the empty string, the encoding is determined using the following rules: • If the original document is an SGML document and the `xml` option is specified, the resulting XML file will use the original encoding if the SGML document has a byte-order mark (an ISO-10646-UCS-2 file) or a special encoding was set using `edit encoding`. If there was no special encoding or it is not an ISO-10646-UCS-2 file, the resulting XML file will use UTF-8 encoding. UTF-8 is the default encoding for XML documents. • If the original document is an SGML document and either no option is specified or the `sgml` option is specified, the resulting SGML file will use the same encoding as the original document. • If the original document is an XML document and the `sgml` option is specified, the resulting SGML file will use the encoding used by the operating system.

	• If the original document is an XML document and either no option is specified or the `xml` option is specified, the resulting XML file will use the same encoding as the original document. XML documents that do not contain an encoding declaration in their header are assumed to have the default XML encoding of UTF-8. `String` *publicId* [optional] If this parameter is not `null` and not an empty string, it is written as the public identifier of the DOCTYPE declaration instead of the original value (if any). If it is `"<none>"` then the PUBLIC identifier will be omitted. `String` *systemId* [optional] If this parameter is not `null` and not an empty string, it is written as the SYSTEM identifier on the DOCTYPE declaration. If it is `"<none>"`, the SYSTEM identifier will be omitted. If this option is `null` or an empty string, the `Application.getOption("writeabsolutesysid")` setting determines how the SYSTEM identifier is written.
Returns	`void`
Throws	AOMException Raised if the method detects any error.

setOption method

This method sets the value of the Arbortext `set` option, scoped to this document.

`setOption`(name, value)	
Parameters	`String` *name* Specifies the option name, which must be a document-scope option. `String` *value* Specifies the new value of the option. Boolean values are specified using the string `on` or `off`.
Returns	`void`
Throws	AOMException Raised if the method detects an error, for example, if `name` is not a valid document-scope option.

undo method

This method reverses the previous change to the document. If called repeatedly, reverses earlier changes.

undo()	
Parameters	None
Returns	void
Throws	AOMException Raised if the method detects an error, for example, when no undo information is available. The associated message gives the reason for the failure.

undoBoundary method

This method inserts a boundary in the undo history so that a subsequent undo will restore changes up to the current state. Normally, the editor inserts a boundary automatically before changes are made by a menu item, toolbar selection, or keyboard shortcut. When implementing a custom application dialog, it may be necessary to call the undoBoundary method before making document changes using the AOM, especially if the dialog is modeless and allows multiple changes to be made which should be undone individually.

The undoBoundary method enables undo history on this document. Normally, a document not associated with a window will not have undo history enabled.

The optional description parameter may be specified to set the label for the Undo menu. Application code can access this label by calling the eval method on the **Acl** interface. For example, in JavaScript:

```
var lbl = Acl.eval("main::undo_label");
```

undoBoundary([description])	
Parameters	String *description* [optional] Specifies the description to use as the Undo menu label for the next undoable change to the document.
Returns	void

undoClear method

Clears the document's undo history. No changes made before this call can be undone.

undoClear( )	
Parameters	None
Returns	void

21

ADocumentEntityEvent interface

The `ADocumentEntityEvent` interface provides specific contextual information associated with the `ADocumentEntityEvent` extension. Use these event types to notify programmers about important document operations related to entities that are not covered by DOM events.

object attribute

Identifies the CMSObject in which the declaration was found.

`object`	
Access	read-only
Returns	`CMSObject`

relatedDocument attribute

The Document in which the declaration was found.

`relatedDocument`	
Access	read-only
Returns	`Document`

relatedNode attribute

DOM Entity containing information about the entity declaration.

`relatedNode`	
Access	read-only
Returns	`Node`

result attribute

A valid entity name to be used a new entity declaration.

`result`	
Access	read-write
Returns	`String`

initADocumentEntityEvent method

Initializes the value of an `ADocumentEntityEvent` created through the `DocumentEntityEvent` interface. You should only call this method before the `ADocumentEntityEvent` has been dispatched using the `dispatchEvent` method, though it can be called multiple times during that phase if necessary. If the `initADocumentEntityEvent` is called multiple times, the final call takes precedence.

`initADocumentEntityEvent`(typeArg, canBubbleArg, cancelableArg, object, relatedDocument, relatedNode, result)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Indicates whether or not the event can bubble. `boolean` *cancelableArg* Indicates whether or not the event's default action can be prevented. `CMSObject` *object* Identifies the CMSObject in which the declaration was found. `Document` *relatedDocument* The Document in which the declaration was found. `Node` *relatedNode* DOM Entity containing information about the entity declaration. `String` *result*
Returns	`void`. A valid entity name to be used a new entity declaration.

22

ADocumentEvent interface

The `ADocumentEvent` interface provides specific contextual information associated with the `ADocumentEvent` extension. Use these event types to notify programmers about important document operations that are not covered by DOM events.

detail attribute

Specifies detail information about the `ADocumentEvent`, depending on the type of event.

`detail`	
Access	read-only
Returns	`long`

relatedDocument attribute

The `relatedDocument` attribute identifies a document related to the event. For `DocumentCreate` event, if the new document is cloned from another document, the `relatedDocument` is the source document that the new document is cloned from.

`relatedDocument`	
Access	read-only
Returns	`Document`

relatedWindow attribute

The `relatedWindow` attribute identifies a window related to the event. For the `DocumentLoad` event, `relatedWindow` is the window that the document loads to. For the `DocumentUnload` event, `relatedWindow` is the window that the document unloads from, as long as the window still exists. If the window is destroyed along with the document, then `relatedWindow` is `null`.

`relatedWindow`	
Access	read-only
Returns	`Window`

targetEncoding attribute

Specifies the encoding in which the document is saved in a `DocumentSaving` event.

`targetEncoding`	
Access	read-only
Returns	`String`

targetURI attribute

Specifies the URI in which the document is saved in a `DocumentSaving` event.

`targetURI`	
Access	read-only
Returns	`String`

initADocumentEvent method

Initializes the value of an `ADocumentEvent` created through the `DocumentEvent` interface. You should only call this method before the `ADocumentEvent` has been dispatched using the `dispatchEvent` method, though it can be called multiple times during that phase if necessary. If the `initADocumentEvent` is called multiple times, the final call takes precedence.

`initADocumentEvent`(typeArg, canBubbleArg, cancelableArg, relatedWindowArg, targetURIArg, targetEncodingArg, detailArg [, relatedDocumentArg])	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Indicates whether or not the event can bubble. `boolean` *cancelableArg* Indicates whether or not the event's default action can be prevented. `Window` *relatedWindowArg* Specifies the `Window` related to the `Event`. `String` *targetURIArg* Specifies the target URI. This value may be `null`. `String` *targetEncodingArg* Specifies the target encoding. This value may be `null`. `long` *detailArg* Specifies the `Event` detail. `Document` *relatedDocumentArg* [optional] Specifies the `Document` related to the `Event`.
Returns	`void`

23

ADocumentType interface

Arbortext extensions to the W3C DOM `DocumentType` interface

doctypeName attribute

If there is an associated DTD or Schema file then this is the basename of that file. For example, if the associated DTD is `axdocbook.dtd`, then this attribute would be `"axdocbook"`.

For a freeform document, this is the local name of the root element.

If the document was structured but the DTD or Schema was not available then this will be the same as the `DocumentType.name` attribute.

If the document was opened as non-structured then this will be `"ascii"`.

`doctypeName`	
Access	read-only
Returns	`String`

doctypeURI attribute

The absolute URI of the document type directory associated with this `DocumentType` or `null` if undefined, for example, for free-form XML or untagged documents.

`doctypeURI`	
Access	read-only
Returns	`String`

tableModels attribute

Returns a list of all the table models valid in a document using this `DocumentType`.

`tableModels`	
Access	read-only
Returns	`StringList`

tableModelCells method

Returns a list containing the name of each tag that is allowed as a cell tag for a table of the specified table model in a document using this `DocumentType`.

tableModelCells(tableModel, header)	
Parameters	String *tableModel* The name of the table model. All of the valid table models in this DocumentType are available using the tableModels attribute. boolean *header* If true, specifies that the list should consist of header cells.
Returns	StringList. A list of all the cell tags for this table model.

tableModelRow method

Returns the name of the tag that is allowed as the row tag for a table of the specified table model in a document using this DocumentType.

tableModelRow(tableModel, header)	
Parameters	String *tableModel* The name of the table model. All of the valid table models in this DocumentType are available using the tableModels attribute. boolean *header* If true, specifies that the name should be for the header row.
Returns	String. The name of the row tag for this table model.

tableModelSupport method

Tests whether a given table model supports a specified feature in documents created using this DocumentType. The same table model may support different features in different documents.

tableModelSupport(tableModel, feature)	
Parameters	String *tableModel* The name of the table model. All of the valid table models in this DocumentType are available using the tableModels attribute. String *feature* The name of the feature to be tested. Valid feature names include "MultipleGrids", "HeaderRows", and "FooterRows".
Returns	True boolean. If the table model supports the feature in this document type.

tableModelTables method

Returns a list containing the name of each tag that is allowed as a table (root) tag for a table of the specified table model in a document using this DocumentType.

tableModelTables(tableModel)	
Parameters	String *tableModel* The name of the table model. All of the valid table models in this DocumentType are available using the tableModels attribute.
Returns	StringList. A list of all the table tags for this table model.

tableModelTableTitle method

Returns the name of the tag that is allowed as the title (or caption) tag for a table of the specified table model in a document using this DocumentType.

tableModelTableTitle(tableModel	
Parameters	String *tableModel* The name of the table model. All of the valid table models in this DocumentType are available using the tableModels attribute.
Returns	String. The name of the title tag for this table model.

tableModelTags method

Returns a list containing the name of all the tags used in the specified table model in a document using this `DocumentType`.

`tableModelTags`(tableModel)	
Parameters	`String` *tableModel* The name of the table model. All of the valid table models in this `DocumentType` are available using the `tableModels` attribute.
Returns	`StringList`. A list of all the tags for this table model.

tableModelWrappers method

Returns a list containing the name of each tag that is allowed as a wrapper tag for a table of the specified table model in a document using this `DocumentType`.

`tableModelWrappers`(tableModel)	
Parameters	`String` *tableModel* The name of the table model. All of the valid table models in this `DocumentType` are available using the `tableModels` attribute.
Returns	`StringList`. A list of all the wrapper tags for this table model.

24

AEditEvent interface

The `AEditEvent` interface provides specific contextual information associated with the `EditEvent` extension. These event types are used to notify programmers of important document operations that are not covered by DOM events.

bufferName attribute

Identifies the name of the paste buffer that is used by the `AOMCut`, `AOMCopy`, or `AOMPaste` event. The standard paste buffer is named default.

`bufferName`	
Access	read-only
Returns	`String`

detail attribute

Identifies detail information about the `Event`, depending on the type of event.

`detail`	
Access	read-only
Returns	`long`

relatedRange attribute

Identifies the `Range` that the event affects.

`relatedRange`	
Access	read-only
Returns	`Range`

initAEditEvent method

Initializes the value of an `AEditEvent` created through the `DocumentEvent` interface. This method should only be called before the `AEditEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initAEditEvent`(typeArg, canBubbleArg, cancelableArg, relatedRangeArg, detailArg [, bufferNameArg])	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `Range` *relatedRangeArg* Specifies the `Range` that is affected by the event. `long` *detailArg* Specifies the `Event` detail. `String` *bufferNameArg* [optional] Specifies the name of the paste buffer that is used by the `AOMCut`, `AOMCopy`, or `AOMPaste` event. The standard paste buffer is named default.
Returns	`void`

25

AElement interface

The Arbortext extension to the W3C DOM `Element` interface.

ATIContentType enumeration

The `tagContentType` attribute has one of the following values.

Note

Some of the content types apply to SGML documents only.

The `ATIContentType` enumeration has the following constants of type `short`.

UNDEFINED_CONTENT = 0
The element is not declared in the DTD.

CDATA_CONTENT = 1
The element is declared as CDATA in the DTD.

RCDATA_CONTENT = 2
The element is declared as RCDATA in the DTD.

EMPTY_CONTENT = 3
The element is declared as EMPTY content in the DTD.

ELEMENT_CONTENT = 4
The element is declared as ELEMENT content in the DTD.

ANY_CONTENT = 5
The element is declared as ANY content in the DTD.

MIXED_CONTENT = 6
The element is declared as MIXED content in the DTD.

tableCell attribute

The `TableCell` associated with this `AElement`. Null if none.

`tableCell`	
Access	read-only
Returns	`TableCell`

tableColumn attribute

The `TableColumn` associated with this `AElement`. Null if none.

`tableColumn`	
Access	read-only
Returns	`TableColumn`

tableGrid attribute

The `TableGrid` associated with this `AElement`. Null if none.

`tableGrid`	
Access	read-only
Returns	`TableGrid`

tableRow attribute

The `TableRow` associated with this `AElement`. Null if none.

`tableRow`	
Access	read-only
Returns	`TableRow`

tableRule attribute

The `TableRule` associated with this `AElement`. Null if none.

`tableRule`	
Access	read-only
Returns	`TableRule`

tableSet attribute

The `TableSet` associated with this `AElement`. Null if none.

`tableSet`	
Access	read-only
Returns	`TableSet`

tagContentType attribute

An integer constant giving the declared content type for the element in the document type. This attribute is deprecated in favor of the contentType attribute in the ElementEditVAL interface which is a W3C standard attribute that returns similar information.

`tagContentType`	
Access	read-only
Returns	`ATIContentType`

getElementsByAttribute method

Returns a `NodeList` of all descendant `Elements` that match the given attribute name and attribute value, in the order in which they are encountered in a pre-order traversal of this `Element` tree.

`getElementsByAttribute`(name, value, selector)	
Parameters	`String` *name* Specifies the name of the attribute to match. The value `"*"` matches all attribute names. `String` *value* Specifies the value of the attribute to match. `ATIElementAttributeSelector` *selector* Specifies how the attribute value should be matched. When selector is `0`, the value parameter is ignored. When selector is `1`, only the elements that match both the name and the value are included.
Returns	`NodeList`. A list of matching element nodes.
Throws	DOMException SYNTAX_ERR: If selector is invalid. INVALID_CHARACTER_ERR: If name is namespace qualified.

getElementsByAttributeNS method

Returns a `NodeList` of all descendant `Elements` that match the given attribute namespace URI, local name, and attribute value, in the order in which they are encountered in a pre-order traversal of this `Element` tree.

`getElementsByAttributeNS`(namespaceURI, localName, value, selector)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to retrieve. The value `"*"` matches all namespaces. `String` *localName* Specifies the local name of the attribute to match. The value `"*"` matches all local attribute names. `String` *value* Specifies the value of the attribute to match. `ATIElementAttributeSelector` *selector*

	Specifies how the attribute value should be matched. When selector is `0`, the value parameter is ignored. When selector is `1`, only the elements that match both the name and the value are included.
Returns	`NodeList`. A list of matching element nodes.
Throws	DOMException SYNTAX_ERR: If selector is invalid. INVALID_CHARACTER_ERR: If localname is namespace qualified.

getInternalAttribute method

Returns the value of an attribute as a string. Allows examination of Arbortext-specific internal attributes, which are not supported using the standard DOM interfaces.

`getInternalAttribute`(name)	
Parameters	`String` *name* Attribute name.
Returns	`String`. Attribute value or `null` if no such attribute is defined.

getInternalAttributes method

Returns a `PropertyMap` containing all attribute names and values.

The list includes Arbortext internal attributes that are excluded from standard DOM processing.

`getInternalAttributes`(includeDefaults)	
Parameters	`boolean` *includeDefaults* If `True`, default attribute values are included.
Returns	`PropertyMap`. Map of attribute name/value pairs.

isTableMarkup method

Returns whether this `Element` is a part of table markup.

isTableMarkup( )	
Parameters	None
Returns	boolean. Returns true if this Element node is part of table markup. Returns false otherwise.

removeInternalAttribute method

Deletes an attribute value. Allows deletion of Arbortext internal attributes which are excluded from standard DOM processing.

removeInternalAttribute(name)	
Parameters	String *name* Name of attribute to delete.
Returns	boolean. Returns true if the attribute was deleted. Returns false otherwise.

setInternalAttribute method

Sets an attribute value. Allows setting of Arbortext internal attributes which are excluded from standard DOM processing.

setInternalAttribute(name, value)	
Parameters	String *name* Name of attribute to set. String *value* New value for attribute.
Returns	boolean. Returns true if a new attribute value was stored. Returns false otherwise.

26

AEvent interface

The Arbortext extension to the W3C DOM `Event` interface. This interface adds the `moduleType` attribute to `Event`, giving the source of the event.

EventDomain enumeration

An integer showing which event domain the event belongs to.

The `EventDomain` enumeration has the following constants of type `unsigned short`.

DOCUMENT_DOMAIN = 1
Shows the event was created by and used in a document.

WINDOW_DOMAIN = 2
Shows the event was created by and used in a window.

APPLICATION_DOMAIN = 3
Shows the event was created by and used in the application.

CMSOBJECT_DOMAIN = 4
Shows the event was created by and used in a CMS object.

CMSSESSION_DOMAIN = 5
Shows the event was created by and used in a CMS session.

CMSADAPTER_DOMAIN = 6
Shows the event was created by and used in a CMS adapter.

EventModule enumeration

An integer showing which event module generated the event.

The `EventModule` enumeration has the following constants of type `unsigned short`.

MUTATION_EVENTS = 1
Shows the event originated from the `MutationEvents` module.

UI_EVENTS = 2
Shows the event originated from the `UIEvents` module.

MOUSE_EVENTS = 3
Shows the event originated from the `MouseEvents` module.

AEDIT_EVENTS = 4
Shows the event originated from the `AEditEvents` module.

WINDOW_EVENTS = 5
Shows the event originated from the `WindowEvents` module.

CONTROL_EVENTS = 6
Shows the event originated from the `ControlEvents` module.

MENU_EVENTS = 7
Shows the event originated from the `MenuEvents` module.

TOOLBAR_EVENTS = 8
Shows the event originated from the `ToolBarEvents` module.

ACTIVEX_EVENTS = 9
Shows the event originated from the `ActivexEvents` module.

ADOCUMENT_EVENTS = 10
Shows the event originated from the `ADocumentEvents` module.

APPLICATION_EVENTS = 11
Shows the event originated from the `ApplicationEvents` module.

CMSOBJECT_EVENTS = 12
Shows the event originated from the `CMSObjectEvents` module.

CMSSESSIONCONSTRUCT_EVENTS = 13
Shows the event originated from the `CMSSessionConstruct` module.

CMSSESSIONCREATE_EVENTS = 14
Shows the event originated from the `CMSSessionCreate` module.

CMSSESSIONFILE_EVENTS = 15
Shows the event originated from the `CMSSessionFile` module.

CMSSESSIONBURSTDOCUMENT_EVENTS = 16
Shows the event originated from the `CMSSessionBurstDocument` module.

CMSSESSIONDISCONNECT_EVENTS = 17
Shows the event originated from the `CMSSessionPreConnect` module.

CMSADAPTERCONNECT_EVENTS = 18
Shows the event originated from the `CMSAdapterConnectEvents` module.

CMSADAPTERDISCONNECT_EVENTS = 19
Shows the event originated from the `CMSAdapterDisconnectEvents` module.

ADOCUMENTENTITY_EVENTS = 20
Shows the event originated from the `ADocumentEntityEvents` module.

domain attribute

The domain identifier of the event.

domain	
Access	read-only
Returns	`unsigned short`

moduleType attribute

The module identifier of the event.

moduleType	
Access	read-only
Returns	`unsigned short`

27

ANode interface

The Arbortext extension to the W3C DOM `Node` interface.

ATIElementAttributeSelector enumeration

Passed as the `selector` parameter to the `getElementsByAttribute` method.

The `ATIElementAttributeSelector` enumeration has the following constants of type `unsigned short`.

ATI_ATTR_ALL_VALUES = 0
Select elements with attributes that match the name parameter only, ignoring the value parameter.

ATI_ATTR_SPECIFIC_VALUE = 1
Select elements with attributes that match both the name and value parameters.

CMSObject attribute

Represents the CMSObject associated with this Node (may be `null`).

This can be accessed for a Document Node or for any other Node type which has an associated OID.

Use the `enclosingObject` attribute on the returned object to obtain any enclosing objects.

`CMSObject`	
Access	read-only
Returns	`CMSObject`
Get throws	CMSException Raised for any errors.

contentModel attribute

Returns the content model as specified in the DTD or schema associated with the owner document of this `Node` as a `DOMString`. The content model has syntax similar to the element definition in a DTD. For example, in a DTD definition, `<!Element book (title | chapter*)>`. The content model for the element book is `(title|chapter*)`.

`contentModel`	
Access	read-only
Returns	`String`

dialog attribute

Returns the `Window` for the XUI dialog associated with this `Node`, if any. This will exist only if there is a DCF file entry associating a XUI dialog with this element in the document.

`dialog`	
Access	read-only
Returns	`Window`

enclosingCell attribute

The table cell this `node` is in, if any.

`enclosingCell`	
Access	read-only
Returns	`TableCell`

enclosingCMSObject attribute

Represents the innermost CMSObject which contains this Node (may be `null`).

Use the `enclosingObject` attribute on the returned object to obtain any enclosing objects.

`enclosingCMSObject`	
Access	read-only
Returns	`CMSObject`
Get throws	CMSException Raised for any errors.

firstOID attribute

A string value that can be spliced into an ACL command or function to indicate the first ACL OID represented by `Node`.

This attribute will normally have the same OID as `lastOID` and will always be the same for `Element` type nodes. They will be different, however, for `Text` nodes that represent more than one text fragment in the document.

`firstOID`	
Access	read-only
Returns	`String`

icon attribute

Used to get or set the name of the display icon that appears to the left of this Node (if any). The icons appear in the Document Map and also appear in the Edit window whenever element tags are set to full or partial display. If this Node has no icon, this returns the string`"None"`.

If an invalid display icon is set, it will act as if it were set to "None".

Note

When using an adapter written using the RAOM (Repository Adapter Object Model), the icons for Nodes representing object boundaries will automatically be managed by the application. This `icon` attribute controls an independent Node icon which can be used by non-RAOM adapters or for other purposes. However, for Nodes representing object boundaries a RAOM adapter **may** override even this `icon` attribute. See the `IOObject.displayIcon` attribute for more information.

The value may be the case-sensitive name of a built-in icon or a full or relative path to a `.bmp` file. When setting this attribute, if a relative path is given, it will be looked for in the search path given by `Application.getOption("graphicspath")`. If the `.bmp` file is not found, it will act as if it were set to the built-in icon “None”.

Note

There is an upper limit (around 200) on the number of **unique** icons that can be set using a `.bmp` file. Once this limit is reached, it will act as if it were set to the built-in icon “None ”.

Built-in icon names are case sensitive and are listed in the following table.

Name	Icon	Default use
Attribute		Indicates that an element has attributes assigned to it.
BadAttribute		Indicates that an element has attributes assigned to it and at least one of those attributes is not legal for the current document type definition.

BadElement		Indicates an element whose position or element name are not legal for the current document type definition.
CheckedOut		Indicates a repository object accessed by a Repository Adapter is locked or checked out by the current user.
CollectionClosed		Indicates that a collection of help topics is hidden from view (in the UNIX help table of contents).
CollectionOpen		Indicates that a collection of help topics is expanded for viewing (in the UNIX help table of contents).
Comment		Indicates that an SGML element is part of a comment Marked Section.
Contracted		Indicates that the element's contents are collapsed and hidden from view.
DataMarkedSection		Indicates that an SGML element is part of a data Marked Section.
DocObject		Indicates a repository object accessed by a Repository Adapter that is neither locked nor checked out.
Document		Indicates a document.
Element		Indicates an SGML or XML element.
Empty		Indicates an element with no content.
End		Indicates an end tag.
Equation		Indicates an equation element.

Expanded		Indicates that the element's contents are expanded for viewing.
FileEntity		Indicates a referenced file entity.
Graphic		Indicates a graphic element.
GrayCheckedOut		Indicates a repository object accessed from a Repository Adapter that is locked or checked out by the current user but is unavailable.
GrayDocObject		Indicates a repository object accessed from a Repository Adapter that is neither locked nor checked out by the current user and is unavailable.
GrayLocked		Indicates a repository object accessed from a Repository Adapter that is locked or checked out by another user and is unavailable.
IgnoreMarkedSection		Indicates that an SGML element is part of an ignored Marked Section.
Locked		Indicates a repository object accessed by a Repository Adapter is locked or checked out by another user.
Missing		Indicates where a required element is missing.
None		No icon displayed.
ReadOnly		Indicates an element that is not editable.
Table		Indicates a table element.

icon	
Access	read-write
Returns	`String`

icon2 attribute

Used to get or set the name of a second display icon that appears to the left of this Node (if any). The icons appear in the Document Map and also appear in the Edit window whenever element tags are set to full or partial display. If this Node has no icon, this returns the string `"None"`.

If an invalid display icon is set, it will act as if it were set to "None".

The value may be the case-sensitive name of a built-in icon or a full or relative path to a `.bmp` file. When setting this attribute, if a relative path is given, it will be looked for in the search path given by `Application.getOption("graphicspath")`. If the `.bmp` file is not found, it will act as if it were set to the built-in icon “None”.

Note

There is an upper limit (around 200) on the number of **unique** icons that can be set using a `.bmp` file. Once this limit is reached, it will act as if it were set to the built-in icon “None”.

Built-in icon names are case sensitive and are listed in the following table.

Name	Icon	Default use
Attribute		Indicates that an element has attributes assigned to it.
BadAttribute		Indicates that an element has attributes assigned to it and at least one of those attributes is not legal for the current document type definition.
BadElement		Indicates an element whose position or element name are not legal for the current document type definition.

CheckedOut		Indicates a repository object accessed by a Repository Adapter is locked or checked out by the current user.
CollectionClosed		Indicates that a collection of help topics is hidden from view (in the UNIX help table of contents).
CollectionOpen		Indicates that a collection of help topics is expanded for viewing (in the UNIX help table of contents).
Comment		Indicates that an SGML element is part of a comment Marked Section.
Contracted		Indicates that the element's contents are collapsed and hidden from view.
DataMarkedSection		Indicates that an SGML element is part of a data Marked Section.
DocObject		Indicates a repository object accessed by a Repository Adapter that is neither locked nor checked out.
Document		Indicates a document.
Element		Indicates an SGML or XML element.
Empty		Indicates an element with no content.
End		Indicates an end tag.
Equation		Indicates an equation element.
Expanded		Indicates that the element's contents are expanded for viewing.

FileEntity		Indicates a referenced file entity.
Graphic		Indicates a graphic element.
GrayCheckedOut		Indicates a repository object accessed from a Repository Adapter that is locked or checked out by the current user but is unavailable.
GrayDocObject		Indicates a repository object accessed from a Repository Adapter that is neither locked nor checked out by the current user and is unavailable.
GrayLocked		Indicates a repository object accessed from a Repository Adapter that is locked or checked out by another user and is unavailable.
IgnoreMarkedSection		Indicates that an SGML element is part of an ignored Marked Section.
Locked		Indicates a repository object accessed by a Repository Adapter is locked or checked out by another user.
Missing		Indicates where a required element is missing.
None		No icon displayed.
ReadOnly		Indicates an element that is not editable.
Table		Indicates a table element.

`icon2`	
Access	read-write
Returns	`String`

lastOID attribute

A string value that can be spliced into an ACL command or function to indicate the last ACL OID represented by `Node`.

This attribute will normally have the same OID as `firstOID` and will always be the same for `Element` type nodes. They will be different, however, for `Text` nodes that represent more than one text fragment in the document.

`lastOID`	
Access	read-only
Returns	`String`

tableNoDelete attribute

Returns `true` if the node is managed by a table model and the table model indicates the `node` should be protected from deletion.

`tableNoDelete`	
Access	read-only
Returns	`boolean`

tableObject attribute

Returns the deepest `table` object (a cell, row, grid, or set) that fully contains the specified `node`. If the specified `node` is not inside table markup, it returns a `null` pointer.

`tableObject`	
Access	read-only
Returns	`TableObject`

userDataKeys attribute

A `DOMStringList` of all keys that have data associated to this node by previous calls to `setUserData`. This is null if no user data exists for the node.

`userDataKeys`	
Access	read-only
Returns	`DOMStringList`

collapse method

Collapses the parent CMS object that contains this node. Can optionally operate on all references to the parent CMS object.

`collapse`(allRefs)	
Parameters	`boolean` *allRefs* Controls whether this operates on all references to the parent CMS object.
Returns	`void`
Throws	CMSException Raised for any errors.

contextPath method

`contextPath` returns a `DOMStringList` that contains possible context paths to make the target `Node` valid at the point indicated by this `Node`. If one of the paths returned is an empty string, the target `Node` can be inserted without any `Node` being added.

`contextPath`(target, depth, maxpaths)	
Parameters	`Node` *target* The `Node` to which the context paths are to be calculated. `unsigned int` *depth* Specifies the maximum tag nesting depth of the paths returned. `unsigned int` *maxpaths* Specifies the maximum number of paths at each depth to return. If depth is 5, and maxpaths is 50, as many as 250 total paths could be returned. If more paths than maxpaths exist at a given depth, only the first maxpaths paths are returned, with no indication that more paths exist.
Returns	`DOMStringList`. Returns the context paths from the point specified by this `Node` to the `target`.

distanceTo method

Finds the distance from this `Node` to another specified `Node`.

`distanceTo` is intended to measure progress through a document in a reasonably linear manner. "Distance" is defined as the number of nodes between the nodes. Such measurements can be used for time estimates, progress dialog boxes, and so on.

If the target node is null, `distanceTo` calculates the distance to the end of the document. If the target node is not null, `distanceTo` calculates the distance to just before the target. Therefore, the sum of the distance between an arbitrary set of targets equals the total document distance, as long as each target is after the previous one.

`distanceTo`(toNode, expandTextEntities, expandFileEntities)	
Parameters	`Node` *toNode* The `Node` to which the distance is to be measured. `boolean` *expandTextEntities* Specifies whether or not text entities should be expanded when measuring the distance. `boolean` *expandFileEntities* Specifies whether or not file entities should be expanded when measuring the distance.
Returns	`long`. Returns the distance between this `Node` and `toNode`. If the distance cannot be calculated, returns `-1`.

expand method

Expands the parent CMS object that contains this node. Can optionally operate on all references to the parent CMS object.

Note

Expanding explicitly forces a collapsed CMS object to be reloaded.

`expand`(allRefs)	
Parameters	`boolean` *allRefs* Controls whether this operates on all references to the parent CMS object.
Returns	`void`
Throws	CMSException Raised for any errors.

getGraphicPath method

If this Node represents a graphic tag, returns the full path (if found) to the referenced graphic.

`getGraphicPath`([makeLocalCopy])	
Parameters	`boolean` *makeLocalCopy* [optional] If `true`, and the referenced graphic resided in a CMS for which there is an active session, the graphic data will be exported to a local temporary file and the full path of the local file will be returned instead of the CMS-specific path (Logical ID).
Returns	`String`. Full file path (if possible) or CMS path (Logical ID) to the referenced graphic. Returns null if a graphic path was not found for this node.
Throws	AOMException Raised for any other errors.

insertTable method

Insert a table as a child of this `Node`.

`insertTable`(tableModel, wrapperTag, colCount, rowCount [, refChild])	
Parameters	`String` *tableModel* The name of the table model for the table to be inserted. Valid table model names are available from the `tableModels` attribute of the `ADocumentType` interface. `String` *wrapperTag* The name of the wrapper tag to insert around the table. Valid wrapper tag names for a given table model in a given document are available from the `tableModelWrappers` method in the `ADocumentType` interface. If a `null` string is passed, the wrapper tag will be chosen randomly. `unsigned long` *colCount* The number of columns in the table. `unsigned long` *rowCount* The number of rows in the table. `Node` *refChild* [optional] The `Node` before which the table is inserted. This must be a child of this node. If this parameter is null, the table will be inserted after the last child of this node.

Returns	`TableSet`. The `TableSet` inserted in the document.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the wrapper tag or table model name is not valid. INVALID_INDEX_ERR: Raised if column or row count is invalid. DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if the node cannot be changed. HIERARCHY_REQUEST_ERR: Raised if a table is not allowed in this context. NOT_FOUND_ERR: Raised if `refChild` is not a child of this node.

setCMSObject method

Associates this Node with the given CMSObject. If this Node is already associated with an object then the new object will be associated with all Nodes in that object's Range. Thus, this might affect other Nodes.

This can be called for a Document or for any other Node type which has an associated OID.

`setCMSObject`(object)	
Parameters	`CMSObject` *object* The new object to associate with this Node.
Returns	`void`
Throws	CMSException Raised for any errors.

28

AOMException exception

Some AOM operations may throw an `AOMException` as specified in their method descriptions. Unlike `DOMException` and other exception interfaces, `AOMException` provides an error message string in the `message` field instead of a numeric code.

Objects that implement the `AOMException` interface include the following property:

`String message`

29

AOMObject interface

The Arbortext `AOMObject` interface is implemented by all AOM and DOM classes.

ObjectType enumeration

The `ObjectType` enumeration is an integer showing which type of object this is.

The `ObjectType` enumeration has the following constants of type `unsigned short`.

NODE_OBJECT = 1
The object is a `Node` object.

RANGE_OBJECT = 2
The object is a `Range` object.

VIEW_OBJECT = 3
The object is a `View` object.

EVENT_OBJECT = 4
The object is a `Event` object.

DOMIMPLEMENTATION_OBJECT = 5
The object is a `DOMImplementation` object.

NODELIST_OBJECT = 6
The object is a `NodeList` object.

NAMEDNODEMAP_OBJECT = 7
The object is a `NamedNodeMap` object.

DOMSTRINGLIST_OBJECT = 8
The object is a `DOMStringList` object.

NAMELIST_OBJECT = 9
The object is a `NameList` object.

XPATHEXPRESSION_OBJECT = 10
The object is a `XPathExpression` object.

XPATHNSRESOLVER_OBJECT = 11
The object is a `XPathNSResolver` object.

XPATHRESULT_OBJECT = 12
The object is a `XPathResult` object.

PROPERTYMAP_OBJECT = 13
The object is a `PropertyMap` object.

STRINGLIST_OBJECT = 14
The object is a `StringList` object.

COMPONENT_OBJECT = 15
The object is a `Component` object.

COMPOSER_OBJECT = 16
The object is a `Composer` object.

TABLEOBJECT_OBJECT = 17
The object is a `TableObject` object.

TABLERECTANGLE_OBJECT = 18
The object is a `TableRectangle` object.

CMSADAPTER_OBJECT = 19
The object is a `CMSAdapter` object.

CMSBROWSEITEM_OBJECT = 20
The object is a `CMSBrowseItem` object.

CMSBROWSEITERATOR_OBJECT = 21
The object is a `CMSBrowseIterator` object.

CMSOBJECT_OBJECT = 22
The object is a `CMSObject` object.

CMSOBJECTLIST_OBJECT = 23
The object is a `CMSObjectList` object.

CMSSESSION_OBJECT = 24
The object is a `CMSSession` object.

IOHOST_OBJECT = 25
The object is a `IOHost` object.

objectType attribute

A code representing the type of the underlying object, as defined by `ObjectType`.

`objectType`	
Access	read-only
Returns	`unsigned short`

30

Application interface

The `Application` interface provides access to Arbortext Editor and Arbortext Publishing Engine global functionality. (That is, features that are not associated with any document, document type, or document component.) It is implemented as a singleton: there is only one `Application` object instantiation in existence.

LoadFlags enumeration

The `LoadFlags` enumerated type is used to construct the `flags` parameter to the `openDocument` method by ORing any of the following options:

The `LoadFlags` enumeration has the following constants of type `int`.

OPEN_RDONLY = 0x0001

Open for read only and do not lock the underlying file. If this is not set, the underlying file will be locked if possible and the document will be read-only if no lock was acquired.

The “checked out” status of CMS Objects will not be affected.

OPEN_DOCRDWR = 0x0002

Open for writing and do not lock the underlying file. The document will be modifiable even though the underlying file is not locked.

If the document was already open in memory, this will additionally attempt to lock the underlying file.

The “checked out” status of CMS Objects will not be affected.

OPEN_NLOCK = 0x0004

Do not lock the underlying file. Overrides all other flags which might acquire a file lock. The resulting document will not be modifiable unless `OPEN_DOCRDWR` is also given.

The “checked out” status of CMS Objects will not be affected.

OPEN_CC = 0x0008

Perform a completeness check when reading the SGML file. This option is ignored for XML documents.

OPEN_NOCC = 0x0010

Suppress the completeness check when reading the SGML file. This option is ignored for XML documents. `OPEN_NOCC` is the default option for SGML documents saved by Arbortext Editor and Arbortext Publishing Engine.

OPEN_NOMSGS = 0x0020

Do not display any parser error messages in a message window. Instead, suppress all warnings and errors.

OPEN_FORCEDT = 0x0040

Use the document type specified by `pubId` and `sysId` to parse the SGML or XML file instead of the document type specified in the file itself.

OPEN_HELPWIN = 0x0080

Open a help document. (Used internally by Arbortext Editor and Arbortext Publishing Engine)

OPEN_XML = 0x0100

Open the document as an XML document even if it does not start with the XML version processing instruction. If not specified, the document is loaded as an SGML document unless the document starts with the XML version header.

OPEN_NOSTYLE = 0x0200

Open the document without loading a style sheet.

OPEN_NODTPROMPT = 0x0400

Do not prompt the user if the document type associated with the document instance does not exist or is not compiled. Instead, return `null`.

OPEN_COMPARE = 0x2000

Open as a specially-treated compare document. (Used internally byArbortext Editor and Arbortext Publishing Engine.)

OPEN_RECTABLES = 0x4000

Cause the table editor to recognize tables immediately after opening the document. By default, table objects are not created until the document is displayed in a window.

OPEN_EDITINIT = 0x8000

Process initialization files immediately after opening the document. This includes sourcing the associated document type instance files (`instance.acl`, `instance.js`, and `instance.vbs`) and the document command files (docname `.acl`, docname`.js`, and docname `.vbs`). By default, these files are not processed until the document is displayed in a window.

OPEN_NEW_DOC = 0x10000

Treat the document as if it were created using the `New` dialog. In this case, the path name is set to null and the document name is of the form `DocumentN`.

OPEN_RECOVERY_PROMPT = 0x20000

Specifies that if an autosave or recovery file exists for the document, the user should be prompted to select the document to open.

OPEN_NAMESPACE_URI = 0x40000

Specifies that the `pubId` parameter is actually a namespace URI instead of a public identifier. If `OPEN_FORCEDT` is also specified, then the namespace URI is used to locate the XML schema to parse the document.

OPEN_FREEFORM = 0x80000

Open the document in free form mode, ignoring the document type specified in the file or by the public identifier `pubId` and system identifier `sysId` parameters.

OPEN_PARSE_STRING = 0x200000
Specifies that the path name parameter `path` is actually a string to parse instead of a file to open. If the string does not contain a DOCTYPE declaration then the `pubId` and or `sysId` parameters must be given so the desired document type is used to parse the string or else `OPEN_FREEFORM` should be specified. If the string contains XML markup but does not start with an XML declaration then `OPEN_XML` must also be specified.

MessageBoxFlags enumeration

The `MessageBoxFlags` enumerated type is used to construct the `flags` parameter to the `messageBox` method by ORing any of the following options:

The `MessageBoxFlags` enumeration has the following constants of type `int`.

MBF_OK = 0x00
Display OK button only. This is the default.

MBF_OKCANCEL = 0x01
Display OK and Cancel buttons.

MBF_ABORTRETRYIGNORE = 0x02
Display Abort, Retry, and Ignore buttons.

MBF_YESNOCANCEL = 0x03
Display Yes, No, and Cancel buttons.

MBF_YESNO = 0x04
Display Yes and No buttons.

MBF_RETRYCANCEL = 0x05
Display Retry and Cancel buttons.

MBF_ICONERROR = 0x10
Display the Error (Stop) icon. This icon is typically used with the Abort, Retry, and Ignore buttons.

MBF_ICONQUESTION = 0x20
Display the Question icon. This icon is typically used with the Yes and No buttons.

MBF_ICONWARNING = 0x30
Display the Warning icon.

MBF_ICONINFORMATION = 0x40
Display the Information icon.

MBF_DEFBUTTON1 = 0x000
The first button is the default. This is the default if no other default button flag is specified.

MBF_DEFBUTTON2 = 0x100
The second button is the default.

MBF_DEFBUTTON3 = 0x200
The third button is the default.

OptionScope enumeration

The `OptionScope` enumerated type is the return type of the `getOptionScope` method, and has the following values:

The `OptionScope` enumeration has the following constants of type `unsigned short`.

INVALID_SCOPE = 0
The option name is invalid.

GLOBAL_SCOPE = 1
The option has global scope.

DOCUMENT_SCOPE = 2
The option has document scope.

WINDOW_SCOPE = 3
The option has window scope.

VIEW_SCOPE = 4
The option has view scope.

acl attribute

The `Acl` global object.

`acl`	
Access	read-only
Returns	`Acl`

activeDocument attribute

A DOM `Document` that represents the Arbortext Editor or Arbortext Publishing Engine active or current document . If the user interface is active, this is the document that has the focus.

activeDocument	
Access	read-only
Returns	Document

activeSession attribute

Represents the active CMSSession (if any).

activeSession	
Access	read-write
Returns	CMSSession

activeWindow attribute

A Window object that represents the Arbortext Editor active window. If the user interface is not active, returns null.

activeWindow	
Access	read-only
Returns	Window

adapterQNames attribute

A list of adapter qualified names for all registered adapters that are available to the application. These values are suitable for use with the Application.getAdapter() method.

adapterQNames	
Access	read-only
Returns	StringList

customProperties attribute

Returns a PropertyMap object containing custom properties for an application. This object is initialized from the application-specific global parameters specified in an application's application.xml file.

customProperties	
Access	read-only
Returns	PropertyMap

documents attribute

A DOM `NodeList` which contains all documents currently opened by Arbortext Editor or Arbortext Publishing Engine. The `NodeList` will be updated as documents are opened and closed.

`documents`	
Access	read-only
Returns	`NodeList`

domImplementation attribute

The `DOMImplementation` object. This is the same value that is returned by a DOM `Document` object's `implementation` attribute.

`domImplementation`	
Access	read-only
Returns	`DOMImplementation`

event attribute

An `Event` object which stores the context of the current event. This attribute can only be obtained from within an event listener.

`event`	
Access	read-only
Returns	`Event`

haveWindows attribute

Returns `true` if the application is running in windows-mode. Returns `false` if running as an Arbortext Publishing Engine server or in one-shot command mode (`-c` specified as a startup option).

`haveWindows`	
Access	read-only
Returns	`boolean`

initDone attribute

Returns `true` if the product has completed initialization.

initDone	
Access	read-only
Returns	boolean

isE3 attribute

Returns `true` if the product is running Arbortext Publishing Engine, either server or interactive mode. Server mode can be determined by also testing the `haveWindows` attribute.

isE3	
Access	read-only
Returns	boolean

lastErrorDetail attribute

Represents the `detail` field of the last exception thrown by the AOM. If the exception had no `detail` field then this will be an empty string. The current value is available only until the next AOM exception is thrown.

This is only available in the COM binding of the `Application` interface because other bindings have direct access to the exception's `detail` field.

lastErrorDetail	
Access	read-only
Returns	String

name attribute

Specifies the name of the Arbortext product, for example, `"Arbortext Editor"`. This string is not localized. The localized version of the string can be obtained by calling `getLocalizedMessage` on the result.

name	
Access	read-only
Returns	String

optionNames attribute

A `StringList` containing the names of all Arbortext `set` options, excluding ACL hook names.

optionNames	
Access	read-only
Returns	StringList

path attribute

specifies the location of the directory that contains the program files needed to run the software.

path	
Access	read-only
Returns	String

userProperties attribute

Returns a PropertyMap object containing user properties (preferences) that override custom properties set for an application. This object is initialized from the user property section of the epic.wcf preferences file. Changes made to the userProperties object are saved back to the preferences file on exit.

userProperties	
Access	read-only
Returns	PropertyMap

alert method

Displays an alert dialog box with the specified message.

alert(message [, title])	
Parameters	String *message* Specifies the message to display in the dialog box String *title* [optional] Specifies the dialog box title. If omitted, the title defaults to "Alert".
Returns	void

confirm method

Displays a modal confirmation dialog box with the specified message.

`confirm`(message [, title])	
Parameters	`String` *message* Specifies the message to display in the dialog box `String` *title* [optional] Specifies the dialog box title. If omitted, the title defaults to `"Confirm"`.
Returns	`boolean`. Returns `true` if the user clicks **OK**. Returns `false` if the user clicks **Cancel**.

constructObject method

Create a new CMSObject for the object referenced by logicalId.

`constructObject`(logicalId [, doc [, reserved]])	
Parameters	`String` *logicalId* Logical ID for a CMS object to be referenced. `Document` *doc* [optional] NULL or document to use for context information. `boolean` *reserved* [optional] This parameter is reserved for future use and should always be `false`.
Returns	`CMSObject`. New object handle.
Throws	CMSException Raised if the CMS object does not exist or an error occurs.

createComposer method

Creates a `Composer` object for the given `ccfPath`.

`createComposer`(ccfPath)	
Parameters	`String` *ccfPath* The path of the CCF file.
Returns	`Composer`. The `Composer` object.
Throws	AOMException Raised if the `ccfPath` is invalid, or if there is an error creating the `Composer`.

createDialogFromDocument method

Creates a dynamic dialog box according to the content of a document.

`createDialogFromDocument`(document [, propertyMap [, parent]])	
Parameters	`Document` *document* The document describing the dialog box. This must conform to the XML User Interface (XUI) document type. `PropertyMap` *propertyMap* [optional] A `PropertyMap` object created by the `createPropertyMap` method to associate with the `Dialog`. This parameter is optional and is not used by Arbortext Editor or Arbortext Publishing Engine. `Window` *parent* [optional] The parent window of the new dynamic dialog. If this parameter is not specified or zero, the parent will be the current active window.
Returns	`Dialog`. The `Dialog` object.

createDialogFromFile method

Creates a dynamic dialog box according to the content of an XML file.

`createDialogFromFile`(filename [, propertyMap [, parent]])	
Parameters	`String` *filename* The XML file containing the dialog box description. This must conform to the XML User Interface (XUI) document type. `PropertyMap` *propertyMap* [optional] A `PropertyMap` object created by the `createPropertyMap` method to associate with the `Dialog`. This parameter is optional and is not used by Arbortext Editor or Arbortext Publishing Engine. `Window` *parent* [optional] The parent window of the new dynamic dialog. If this parameter is not specified or zero, the parent will be the current active window.
Returns	`Dialog`. The `Dialog` object.

createEvent method

Creates an event of type `ApplicationEvent`.

createEvent(eventType)	
Parameters	String *eventType* The eventType parameter specifies the type of Event interface to be created. If the Event interface specified is supported by the implementation this method will return a new Event of the interface type requested. If the Event is to be dispatched via the dispatchEvent method the appropriate event init method must be called after creation in order to initialize the Event's values.
Returns	Event. The newly created Event
Throws	AOMException Raised if the implementation does not support the type of Event interface requested.

createPropertyMap method

Creates an empty PropertyMap object that is an unordered collection of name-value pairs.

createPropertyMap( )	
Parameters	None
Returns	PropertyMap. The PropertyMap object.

createScriptContext method

Creates a ScriptContext object that may be used to load, compile, and execute scripts using the Microsoft Windows Script engine. This method is only available in the COM binding of the Application interface.

createScriptContext(language, name)	
Parameters	String *language* Specifies the name of the Microsoft Windows Script language to initialize the script context. The name must be either "VBScript" or "JScript", the ProgID values of the respective script language. A ProgID is the version-independent, user-friendly name of the GUID (Globally Unique Identifier) found in the Windows registry. Any ActiveScript-compatible script language can be named and used, but only VBScript and JScript are supported. String *name* Specifies the name of the script.
Returns	ScriptContext. The IDispatch pointer to the ScriptContext object. If the object creation fails, the method returns null.

createStringList method

Creates an empty StringList object that is an ordered collection of DOMStrings.

createStringList(size)	
Parameters	long *size* The initial size of the array.
Returns	StringList. The StringList object.

createTableObjectStore method

Creates an empty TableObjectStore object that is a collection of TableObjects.

createTableObjectStore( )	
Parameters	None
Returns	TableObjectStore. The TableObjectStore object.

createTableTilePlex method

Creates an empty TableTilePlex object which can represent a table selection in a document.

createTableTilePlex( )	
Parameters	None
Returns	TableTilePlex. The TableTilePlex object.

createWindow method

Creates a window of the specified windowClass with optional components given by flags. The window created is not initially displayed. Use the Window.show() method to make the window appear.

createWindow(windowClass [, flags [, doc [, geometry [, parent [, xuiPath]]]]])	
Parameters	String *windowClass* Windows are of two types: • **Document** class windows that display a document tree and have a windowClass of edit, helpwin[1-4], or msgwin[1-4]. • **Dialog** class windows that display either a list selection dialog box of windowClass list or xui. The class determines the default geometry and, for classes other than list, the class-specific keymap. By default, a new keymap is created for the window on the first **map** ACL command processed for the window. This keymap has the name windown, where n is the identifier of the window, and is deleted when the window is destroyed. If a **set keymap=user** ACL command is executed for the window, or if bit 0x00040 is specified in flags, the global keymap for the class will be used. See the flags bit descriptions below. . int *flags* [optional] The flags parameter is a bit mask that depends on the windowClass and is defined below: The following are the flag bits for list and xui class windows. • 0x1 - Supply vertical scrollbar. • 0x2 - Verify input (that is, set verify Item attribute). • 0x4 - Supply an **Apply** button, • 0x8 - Supply a **Help** button. • 0x10 - For XUI dialog boxes, delete the document

	after the window is destroyed. In the following example, `doc` will be destroyed automatically after `win` is destroyed:
	`Document doc = Application.openDocument("c:\\myproject\\myxuifile", 1);`
	`Window win = Application.createWindow("xui", 0x10, doc);`

	• `0x040000` - Make the new window a top-level window with its parent being the desktop. This flag is only useful to the xui and list class windows; the windows of all other window classes are created as top-level windows. **OK** and **Cancel** buttons are always supplied. The following are the `flag` bits for document class (all non-list) windows. • `0x00001` - Supply vertical scrollbar (pane). • `0x00002` - Supply menu bar. • `0x00004` - Supply command subwindow if `windowClass` is edit. • `0x00008` - Supply message footer subwindow. • `0x00010` - Automatically call `ADocument.close()` on the attached document when the window is destroyed. (pane). • `0x00020` -Supply edit toolbar (that is, Toolbar 1) (pane). • `0x00040` -Attach the global window class keymap to the window instead of creating a private keymap (pane). • `0x00080` - Supply horizontal scrollbar (pane). • `0x00100` - Do **edit** command intializations, include reading the document type instance command files (`instance.acl` and `instance.js`) and document command files (docname`.acl` and docname `.js`) if they exist, and calling the ACL `editfilehook` when a document is attached to the window. This bit applies only to edit class windows (pane). • `0x00800` - Make the window a typical user edit window (as opposed to a display window). • `0x01000` - Supply a table column width ruler (pane). • `0x02000` - Supply a table row height ruler (pane). • `0x04000` - Supply the Markup toolbar (that is, Toolbar 2). • `0x08000` - Supply the Table toolbar (that is, Toolbar 3).

	• `0x10000` - Supply the Application toolbar (that is, Toolbar 4).

	• `0x080000` - Do not update the Arbortext Editor **File** menu list of recently edited documents, or the Microsoft Windows list of recently edited documents with the path name associated with the `doc` parameter (if the `0x00800` flag is specified). Note that this does not apply to documents subsequently loaded in the window. If a menu bar is requested, it must be initialized using the **menu_load** or **menu_add** ACL commands before the window is first displayed. If a message footer is created, error messages and output from the **message** ACL command are displayed in the left part of the footer if the message is short enough (otherwise a popup dialog box is used). Any messages directed to the message footer are considered transient and are erased on the next key or button event received in the window. . `Document` *doc* [optional] Specifies the document tree to be attached to the window. The document must not already be displayed in another window. If it is, the function returns `null`. If `doc` is `null`, a scratch document is created that will automatically be destroyed when the window is destroyed. In this case, the associated document type is ascii for edit class windows or the built-in help document type for other classes. This parameter does not apply to list class windows and is ignored if given. . `String` *geometry* [optional] Specifies the initial geometry for the window and is a string of the form `WxH+X+Y`, where `W` and `H` are the width and height of the window in pixels, and `X` and `Y` give the location of the upper left corner of the window. . `Window` *parent* [optional] An optional parameter used to specify the parent window for the new window. Only supports dialog class. . `String` *xuiPath*

	[optional] An optional parameter used only by edit windows to supply an alternative XUI file to define the toolbars used by the edit window. If `xuiPath` is not supplied (or empty), then Arbortext-path `\lib\dialogs\editwindow.xml` is used.
Returns	`Window`. A new `Window` object.
Throws	AOMException Raised if the method detects any error.

error method

Sounds a beep and displays the error message specified by `message` in the status bar of the active window if possible, otherwise in a separate dialog. The message is also assigned to the `ERROR` predefined ACL variable

The `error` method is used by Arbortext Editor to display most error messages.

`error`(message)	
Parameters	`String` *message* Specifies the message to display.
Returns	`void`

getAdapter method

Returns the requested adapter if available.

`getAdapter`(adapterQName)	
Parameters	`String` *adapterQName* Adapter qualified name.
Returns	`CMSAdapter`. The requested `CMSAdapter` or `null` if no such adapter is registered.

getCustomDirectory method

Returns the installation directory for a specified application. If `name` is omitted or the null string, then the default custom directory is returned, either the first value of the `APTCUSTOM` environment variable if set or else the `custom` subdirectory in the product installation directory.

If `name` is a number, then this specifies the 0-based index into the list of custom directories. This allows an iterator to enumerate the list of custom directories by calling this method in a loop, incrementing the index until a null string is returned.

If `name` is a negative integer, then the list is traversed in reverse. `"-1"` returns the last custom directory, `"-2"` the second to last custom directory and so on.

`getCustomDirectory`([name])	
Parameters	`String` *name* [optional] Specifies the application name or index.
Returns	`String`. The full path name of the specified application's custom directory or `null` if `name` is not a loaded application name or if it specifies an index out of range.

getLocale method

Returns the requested locale string.

getLocale([category])	
Parameters	String *category* null[optional] or a supported locale category string. On UNIX, this returns the shorter form of the locale name regardless of the value of the category parameter. On Windows, the following category strings are supported (case sensitive). category Method Result null or empty string Abbreviated locale string (2-3 letter code). Example: ENU LC_COLLATE The locale governing certain collating functions. LC_CTYPE The locale governing human-readable messages. LC_MESSAGES Equivalent to LC_CTYPE. LC_MONETARY The locale governing money formatting. LC_NUMERIC The locale governing numeric formatting. LC_TIME The locale governing time formatting. Any other string. Method will return an empty string.
Returns	String. The requested locale string. Will be null for any unrecognized category string.

getLocalizedMessage method

Returns the localized version of the specified message from the default message catalog file.

getLocalizedMessage(message)	
Parameters	String *message* The message to localize.
Returns	String. The localized version of message. If the message is not found in the message file, returns message.

getOption method

Returns the value of the Arbortext set option, in global scope.

getOption(name)	
Parameters	String *name* Specifies the option name.
Returns	String. The string value of the option, or null if name is not a valid option name. Boolean values return on or off.

getOptionScope method

Returns the scope of the Arbortext set option.

getOptionScope(name)	
Parameters	String *name* Specifies the option name.
Returns	OptionScope. A code representing the scope of the option as defined by OptionScopeType. If name is not a valid option name, returns INVALID_SCOPE.

getScriptContext method

Returns an IDispatch pointer to a ScriptContext object for the running script specified by the name parameter. This method is only available in the COM binding of the Application interface.

getScriptContext(name)	
Parameters	String *name* Specifies the name of the running script. The script name is not the file name. It is one of several possible names: the name passed to CreateScriptContext, a constructed name that is unique to the dialog for the script context in a

	XUI dialog, "EpicJS" for the global JScript context, or "EpicVBS" for the global VBScript context.
Returns	`ScriptContext`. The `ScriptContext` object or `null` if the named script does not exist.
Throws	AOMException Raised if Active Scripting is not supported in this version of Arbortext Editor.

logicalIdExists method

Tests the existence of Logical IDs associated with any active CMS session as well as for file-system and http/https resources.

`logicalIdExists`(logicalId)	
Parameters	`String` *logicalId* Logical ID for a CMS or a file-system or http/https resource.
Returns	`trueboolean`. if the referenced object/resource exists, `false` otherwise. If there is any error accessing the resource, this will return `false` and will not throw an exception.

logicalIdToSession method

If the specified path is the correct Logical ID format for a connected CMS, this returns the CMSSession object associated with that session.

`logicalIdToSession`(logicalId)	
Parameters	`String` *logicalId* CMS-specific Logical ID. The existence of this Logical ID is not considered when looking for a session.
Returns	`CMSSession`. CMSSession which claimed ownership of the given Logical ID. Will return `null` if no session claimed ownership.

messageBox method

Displays a message box with the text `message` and optional title `title`. The `flags` parameter determines what predefined buttons and icons display in the message box, and is formed by ORing the flags from the following groups of flag bits.

Specify one of the following flags to indicate the buttons that will display in the message box:

- 0x00 — Display OK button only. This is the default.
- 0x01 — Display OK and Cancel buttons.
- 0x02 — Display Abort, Retry, and Ignore buttons.
- 0x03 — Display Yes, No, and Cancel buttons.
- 0x04 — Display Yes and No buttons.
- 0x05 — Display Retry and Cancel buttons.

Specify one of the following flags to indicate the icon to display in the message box. If you do not specify one of these flags, an icon does not display.

- 0x10 — Display the Error (Stop) icon. This icon is typically used with the Abort, Retry, and Ignore buttons
- 0x20 — Display the Question icon. This icon is typically used with the Yes and No buttons.
- 0x30 — Display the Warning icon.
- 0x40 — Display the Information icon.

Specify one of the following flags to indicate the default button:

- 0x000 — The first button is the default. This is the default if no other default button flag is specified.
- 0x100 — The second button is the default.
- 0x200 — The third button is the default.

If the dialog box has a `Cancel` or `Ignore` button, the function returns `3` if the `Cancel` or `Ignore` button or `ESC` key was pressed, or if the dialog box was closed from the `Close` system menu or `Close` button. If the dialog box does not have a `Cancel` or `Ignore` button and is closed with the `ESC` key or by the `Close` system menu or `Close` button, the function returns `2` if the dialog has only `Yes` and `No` buttons. If the dialog box only has an `OK` button, it returns a `1`.

messageBox(message [, flags [, title]])	
Parameters	String *message* Specifies the message to display. int *flags* [optional] Specifies a bitmask of options constructed by ORing the bits from the MessageBoxFlags enumeration. String *title* [optional] Specifies the dialog box title. If omitted, the title defaults to "Message".
Returns	int. The return value is one of the following: • 1 - The first button (Yes, OK, Abort, or Retry) was pressed. • 2 - The No button was pressed. • 3 - The third button (Cancel or Ignore) was pressed.

openDocument method

Reads an XML or SGML file and creates a new Document object that may be used to navigate the document's content. The method may also be used to create an empty document if path is null, similar to the createDocument method of the DOMImplementation interface.

The pubid and sysid arguments specify the document type for the document if path is omitted or null, if the associated file does not specify a DOCTYPE declaration, or if bit OPEN_FORCEDT is included in flags. The pubid and sysid arguments are ignored if path specifies an SGML file that starts with a DOCTYPE declaration, if OPEN_FORCEDT is not specified, or if path specifies a binary document file. If the document type is not specified, is "ascii", or cannot be determined the document is opened in untagged mode.

openDocument([path [, flags [, name [, pubId [, sysId [, stylesheet]]]]]])	
Parameters	String *path* [optional] Specifies the path name of a document directory or the file name from which to load the initial contents of the document tree. May be a "file://" or "http://" URL. If the URL specifies a server supporting WebDAV, the file will be opened for editing (Windows only). If the server does not support WebDAV, the file will be opened as read-only. If null or an empty string, the document is empty. int *flags*

	[optional] A bitmask that specifies open options. Constructed by ORing the bits from the `LoadFlags` enumeration. `String` *name* [optional] Specifies a name to be used for informational purposes. If `null` or the empty string, the base name of `path` is used. If `path` is `null` or empty, an internal name is assigned. `String` *pubId* [optional] Specifies the public identifier of the document type. `String` *sysId* [optional] Specifies the system identifier of the document type. `String` *stylesheet* [optional] Specifies the style sheet to be used instead of the default style sheet for the document. If flag `OPEN_NOSTYLE` is set, this parameter is ignored. If the specified style sheet does not exist, an exception is raised.
Returns	`Document`. A new `Document` object.
Throws	AOMException Raised if the method detects any error.

print method

Outputs a string to the message window. If the user interface is not open on Windows, the message is discarded. In Arbortext Publishing Engine on Windows, the message is sent to the trace window if it is open, otherwise it is discarded.

`print`([str])	
Parameters	`String` *str* [optional] Specifies the string to print. A line break is not added. Newline characters in the string will cause line breaks. If the parameter is omitted or `null`, a line break is output. This method can not be used from Visual Basic since `print` is a reserved word in Visual Basic and can't be used as a method name on any object.
Returns	`void`

prompt method

Displays a modal dialog box with the specified message `prompt`, a text input field, and **OK** and **Cancel** buttons.

`prompt`(prompt [, value [, title]])	
Parameters	`String` *prompt* Specifies the message to display in the dialog box `String` *value* [optional] Specifies the initial value displayed in the text input field. `String` *title* [optional] Specifies the dialog box title. If omitted, the title defaults to `"Prompt"`.
Returns	`String`. Returns the string in the text input field if the user clicks **OK**. Returns `null` if the user clicks **Cancel**.

quit method

Terminates the application with the exit status `status`. The parameter `code` determines if the user is prompted for unsaved changes or not and has one of the values:

- `0` — prompt about any unsaved changes.
- `1` — save all modified documents without prompting.
- `2` — do not prompt about unsaved changes and quit without saving modified documents.

`quit`([code [, status]])	
Parameters	`int` *code* [optional] Specifies whether unsaved changes are prompted (0), saved without prompting (1), or discarded without prompting (2). The default is 0. `int` *status* [optional] Specifies the exit status for the program. The default is 0.
Returns	`void`

registerIOAdapter method

Called during startup by the adapter to register itself with Arbortext Editor. This call should be the last thing done in the initialization/loading code for the adapter. An adapter cannot be unregistered once it has been registered.

`registerIOAdapter`(adapter, name, qName)	
Parameters	`IOAdapter` *adapter* The adapter instance. `String` *name* The human-readable name of the adapter. `String` *qName* The qualified name that uniquely identifies this adapter. Qualified names should follow the same reverse domain name convention used by Java.
Returns	`void`
Throws	CMSException Raised for any error.

run method

Runs the macro or alias named `name`. The name is first looked up as a macro using the active document macro scope. If no such macro is found in any scope, then `name` is looked up as a command alias.

`run`(name)	
Parameters	`String` *name* The name of the macro or alias to execute.
Returns	`void`
Throws	AOMException Raised if `name` is not recognized as a macro or alias or if an error occurs while executing the macro or alias.

setOption method

Sets the value of the Arbortext `set` option, in global scope. If `name` specifies a `Document`- or `View`-scoped option, setting the value does not affect any existing documents or views, only newly created objects.

`setOption`(name, value)	
Parameters	`String` *name* Specifies the option name. `String` *value*

	Specifies the new value of the option. Boolean values are specified using the string `on` or `off`.
Returns	`void`
Throws	AOMException Raised if the method detects an error (for example, if `name` is not a valid option).

31

ApplicationEvent interface

The `ApplicationEvent` interface provides specific contextual information associated with the `ApplicationEvent`.

detail attribute

Specifies detail information about the `ApplicationEvent` , depending on the type of event.

`detail`	
Access	read-only
Returns	`long`

initApplicationEvent method

Initializes the value of an `ApplicationEvent` created through the `Application` interface. This method should only be called before the `ApplicationEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initApplicationEvent`(typeArg, canBubbleArg, cancelableArg, detailArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `long` *detailArg* Specifies the `Event` detail.
Returns	`void`

32

ARange interface

The Arbortext extension to the W3C DOM `Range` interface.

`ARange` adds four read-only attributes (`startOID`, `startPos`, `endOID`, `endPos`) that give the start and end points of the Range as strings that may be spliced into ACL commands. Note that ACL represents a point as an OID/POS pair.

Arbortext Editor (and Arbortext Publishing Engine) and the DOM represent ranges differently. Therefore, the individual components of a DOM range endpoint (attributes `startNode`, `startOffset`) and an Arbortext endpoint (attributes `startOID`, `startPos`) may differ. That is, the OID indicated by `startOID` will not necessarily be the starting OID for the node indicated by `startNode`, and the integer value `startOffset` will not necessarily be equal to the integer value `startPos`. Nor will there necessarily be equivalences between `endNode` and `endOID` or `endOffset` and `endPos`.

PTC only guarantees that the point in the document represented by the pair (`startNode`, `startOffset`) will be the same point as that indicated by the pair (`startOID`, `startPos`) and that the point represented by the pair (`endNode`, `endOffset`) will be the same point as that represented by the pair (`endOID`, `endPos`).

The DOM allows the endpoint of a range to be within a processing instruction; Arbortext products do not. If a DOM (node, offset) pair is located within a processing instruction, the corresponding (OID, pos) pair will indicate the point just before the start of the processing instruction (if the [node, offset] is the start of the range) or just after the end of the processing instruction (if the [node, offset] is the end of the range).

MarkupFlags enumeration

The `MarkupFlags` enumerated type is used to construct the `flags` parameter to the `toMarkupStringEx` method by ORing any of the following options:

The `MarkupFlags` enumeration has the following constants of type `int`.

MARKUP_HEADER = 0x01
Include the XML or SGML header associated with the `Range` . If the `Range` does not include the entire document, this will be a fragment header.

MARKUP_FORCE_XML = 0x02
Use XML syntax in the string returned even if the `Range` is in an SGML document.

MARKUP_FORCE_SGML = 0x04
Use SGML syntax in the string returned even if the `Range` is in an XML document.

MARKUP_NO_PI = 0x08
Suppress Arbortext processing instructions. Arbortext processing instructions can also be suppressed using the `writepi` set option.

MARKUP_FORCE_PI = 0x10
Force Arbortext processing instructions to be included. This option overrides the `MARKUP_NO_PI` option and the `writepi` set option.

MARKUP_EXPAND_XINCLUDE = 0x20
Force XML inclusions to be replaced by their contents.

MARKUP_CHAR = 0x40
Non-ASCII characters are converted according to the current `writenonasciichar` set option. If the `entityoutputconvert` set option is also `on`, then character entities will also be output according to the `writenonasciichar` set option.

allowedInsertElements attribute

Elements that can be inserted into the Document or DocumentFragment at the start of the `Range` such that the result will be compliant with VAL_SCHEMA validity type. If the start container is a Text node it will be assumed to be split into two text nodes and the list of elements valid between them will be returned.

`allowedInsertElements`	
Access	read-only
Returns	`NameList`

allowedSurroundElements attribute

Elements that can surround the `Range` such that the result will be compliant with VAL_SCHEMA validity type.

`allowedSurroundElements`	
Access	read-only
Returns	`NameList`

contextString attribute

This function returns a `DOMString` describing the context of the start of this `Range`. This string consists of a list of element names and parentheses, such as: `doc(body(chapter(title()para0(title()para(` The left parenthesis following an element name represents a start tag, and the right parenthesis represents the end tag for the corresponding unmatched start tag. If this `Range` is before the opening start tag or if context checking is not relevant for the current document, a null string will be returned.

`contextString`	
Access	read-only
Returns	`String`

endOID attribute

The end OID of the `Range`. Note that the OID indicated by the `endOID` is not necessarily the same as the ending OID for the node indicated by the `endNode`.

`endOID`	
Access	read-only
Returns	`String`
Get throws	DOMException INVALID_STATE_ERR: Raised if the `Range` has already been detached.

endPos attribute

The end position (in ACL) of the `Range`. Note that the position indicated by the `endPos` is not necessarily equal to the value of `endOffset`.

`endPos`	
Access	read-only

Returns	`String`
Get throws	DOMException INVALID_STATE_ERR: Raised if the `Range` has already been detached.

startOID attribute

The start OID of the `Range`. Note that the OID indicated by the `startOID` is not necessarily the same as the starting OID for the node indicated by the `startNode`.

`startOID`	
Access	read-only
Returns	`String`
Get throws	DOMException INVALID_STATE_ERR: Raised if the `Range` has already been detached.

canInsertNode method

This method indicates whether a `Node` can be inserted at a position specified by the start of this `Range` such that the result is compliant with VAL_SCHEMA validity type. If the container is a text node, it will be considered to have been split and the test will be made between the two resulting text nodes.

`canInsertNode`(node)	
Parameters	`Node` *node* The `Node` to be inserted.
Returns	`unsigned short`. A validation state constant.

canInsertNodeWithFixup method

This method indicates whether a `Node` can be inserted at a position specified by the start of this `Range` such that the result is compliant with VAL_SCHEMA validity type. This test considers adding required ancestors or descendents to make context valid.

`canInsertNodeWithFixup`(node)	
Parameters	`Node` *node* The `Node` to be inserted.
Returns	`unsigned short`. A validation state constant.

insertNodeWithFixup method

This method inserts a `Node` to the position specified by the start of this `Range`. It will try to add required ancestors or descendents to make context compliant with VAL_SCHEMA validity type. If the start container of the range is a text node it will be split and the node will be inserted between the two resulting text nodes.

`insertNodeWithFixup`(node)	
Parameters	`Node` *node* The `Node` to be inserted.
Returns	`Range`. The `Range` inserted.
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if an ancestor container of the start of the Range is read-only. WRONG_DOCUMENT_ERR: Raised if `newNode` and the container of the start of the Range were not created from the same document. HIERARCHY_REQUEST_ERR: Raised if the container of the start of the Range is of a type that does not allow children of the type of `newNode` or if `newNode` is an ancestor of the container. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. RangeException INVALID_NODE_TYPE_ERR: Raised if `newNode` is an Attr, Entity, Notation, or Document node.

insertParsedString method

Parses `text` and inserts the resulting DOM objects into a document at the location indicated by the start of the `Range`.

`insertParsedString`(text)	
Parameters	`String` *text* The text to be inserted. Markup is interpreted as XML or SGML according to the target document. If an empty string, this method does nothing.

Returns	`void`
Throws	AOMException Raised if the method detects an error, for example, the insertion is not permitted due to context checking.

toMarkupString method

Returns the contents of a `Range` as a string. This string contains the character data and markup representing the entire contents of the range.

`toMarkupString`( )	
Parameters	None
Returns	`String`. The contents of the `Range`.
Throws	DOMException INVALID_STATE_ERR: Raised if the `Range` has already been detached.

toMarkupStringEx method

Returns the contents of a `Range` as a string, with control over the markup. This string contains the character data and markup representing the entire contents of the range.

`toMarkupStringEx`([flags])	
Parameters	`int` *flags* [optional] A bitmask that specifies markup options. Constructed by ORing the bits from the `MarkupFlags` enumeration.
Returns	`String`. The contents of the `Range`.
Throws	DOMException INVALID_STATE_ERR: Raised if the `Range` has already been detached.

33

W3C Attr interface

The `Attr` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

The attribute's effective value is determined as follows: if this attribute has been explicitly assigned any value, that value is the attribute's effective value; otherwise, if there is a declaration for this attribute, and that declaration includes a default value, then that default value is the attribute's effective value; otherwise,

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

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

isId attribute

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

- If validation occurred using an XML Schema [XML Schema Part 1] while loading the document or while invoking `Document.normalizeDocument()`, the post-schema-validation infoset contributions (PSVI contributions) values are used to determine if this attribute is a schema-determined ID attribute using the schema-determined ID definition in [XPointer].
- If validation occurred using a DTD while loading the document or while invoking `Document.normalizeDocument()`, the infoset **[type definition]** value is used to determine if this attribute is a DTD-determined ID attribute using the DTD-determined ID definition in [XPointer].
- from the use of the methods `Element.setIdAttribute()`, `Element.setIdAttributeNS()`, or `Element.setIdAttributeNode()`, i.e. it is an user-determined ID attribute;

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

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

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

`isId`	
Access	read-only
Returns	`boolean`

name attribute

Returns the name of this attribute.

`name`	
Access	read-only
Returns	`String`

ownerElement attribute

The `Element` node this attribute is attached to or `null` if this attribute is not in use.

`ownerElement`	
Access	read-only
Returns	`Element`

schemaTypeInfo attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

The type information associated with this attribute. While the type information contained in this attribute is guarantee to be correct after loading the document or invoking `Document.normalizeDocument()`, `schemaTypeInfo` may not be reliable if the node was moved.

`schemaTypeInfo`	
Access	read-only
Returns	`TypeInfo`

specified attribute

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

In summary:

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

`specified`	
Access	read-only
Returns	`boolean`

value attribute

On retrieval, the value of the attribute is returned as a string. Character and general entity references are replaced with their values. See also the method `getAttribute` on the `Element` interface.

On setting, this creates a `Text` node with the unparsed contents of the string. I.e. any characters that an XML processor would recognize as markup are instead treated as literal text. See also the method `setAttribute` on the `Element` interface.

`value`	
Access	read-write
Returns	`String`
Set throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

34

W3C CDATASection interface

The `CDATASection` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

The `CDATASection` interface inherits from the `CharacterData` interface through the `Text` interface. Adjacent `CDATASection` nodes are not merged by use of the `normalize` method of the `Node` interface.

Note

Because no markup is recognized within a `CDATASection`, character numeric references cannot be used as an escape mechanism when serializing. Therefore, action needs to be taken when serializing a `CDATASection` with a character encoding where some of the contained characters cannot be represented. Failure to do so would not produce well-formed XML.

One potential solution in the serialization process is to end the CDATA section before the character, output the character using a character reference or entity reference, and open a new CDATA section for any further characters in the text node. Note, however, that some code conversion libraries at the time of writing do not return an error or exception when a character is missing from the encoding, making the task of ensuring that data is not corrupted on serialization more difficult.

35

W3C CharacterData interface

The `CharacterData` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

data attribute

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

`data`	
Access	read-write
Returns	`String`
Get throws	DOMException DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` variable on the implementation platform.
Set throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

length attribute

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

`length`	
Access	read-only
Returns	`unsigned long`

appendData method

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

`appendData`(arg)	
Parameters	`String` *arg* The `DOMString` to append.

Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

deleteData method

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

`deleteData`(offset, count)	
Parameters	`unsigned long` *offset* The offset from which to start removing. `unsigned long` *count* The number of 16-bit units to delete. If the sum of `offset` and `count` exceeds `length` then all 16-bit units from `offset` to the end of the data are deleted.
Returns	`void`
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

insertData method

Insert a string at the specified 16-bit unit offset.

`insertData`(offset, arg)	
Parameters	`unsigned long` *offset* The character offset at which to insert. `String` *arg* The `DOMString` to insert.

Returns	`void`
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

replaceData method

Replace the characters starting at the specified 16-bit unit offset with the specified string.

`replaceData`(offset, count, arg)	
Parameters	`unsigned long` *offset* The offset from which to start replacing. `unsigned long` *count* The number of 16-bit units to replace. If the sum of `offset` and `count` exceeds `length`, then all 16-bit units to the end of the data are replaced; (i.e., the effect is the same as a `remove` method call with the same range, followed by an `append` method invocation). `String` *arg* The `DOMString` with which the range must be replaced.
Returns	`void`
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

substringData method

Extracts a range of data from the node.

`substringData`(offset, count)	
Parameters	`unsigned long` *offset*

	Start offset of substring to extract. `unsigned long` *count* The number of 16-bit units to extract.
Returns	`String`. The specified substring. If the sum of `offset` and `count` exceeds the `length`, then all 16-bit units to the end of the data are returned.
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified `offset` is negative or greater than the number of 16-bit units in `data`, or if the specified `count` is negative. DOMSTRING_SIZE_ERR: Raised if the specified range of text does not fit into a `DOMString`.

36

W3C CharacterDataEditVAL interface

The `CharacterDataEditVAL` interface is defined in the W3C Document Object Model (DOM) Level 3 Validation Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Val.)

This interface extends the `NodeEditVAL` interface with additional methods for document editing. An object implementing this interface must also implement `CharacterData` interface. When validating `CharacterData` nodes, the NodeEditVAL.nodeValidity operation must find the nearest parent node in order to do this; if no parent node is found, VAL_UNKNOWN is returned. In addition, when VAL_INCOMPLETE is passed in as an argument to the `NodeEditVAL.nodeValidity` operation to operate on such nodes, the operation considers all the text and not just some of it.

canAppendData method

Determines if character data can be appended.

`canAppendData`(arg)	
Parameters	`String` *arg* Data to be appended.
Returns	`unsigned short`. A validation state constant.

canDeleteData method

Determines if character data can be deleted.

`canDeleteData`(offset, count)	
Parameters	`unsigned long` *offset* Offset. `unsigned long` *count* Number of 16-bit units to delete.
Returns	`unsigned short`. A validation state constant.
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in data, or if the specified count is negative.

canInsertData method

Determines if character data can be inserted.

`canInsertData`(offset, arg)	
Parameters	`unsigned long` *offset* Offset. `String` *arg* Argument to be set.
Returns	`unsigned short`. A validation state constant.
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in data.

canReplaceData method

Determines if character data can be replaced.

`canReplaceData`(offset, count, arg)	
Parameters	`unsigned long` *offset* Offset. `unsigned long` *count* Replacement. `String` *arg* Argument to be set.
Returns	`unsigned short`. A validation state constant.
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in data, or if the specified count is negative.

canSetData method

Determines if character data can be set.

`canSetData`(arg)	
Parameters	`String` *arg* Argument to be set.
Returns	`unsigned short`. A validation state constant.

isWhitespaceOnly method

Determines if character data is only whitespace.

`isWhitespaceOnly`( )	
Parameters	None
Returns	`unsigned short`. A validation state constant.

37

CMSAdapter interface

Represents an installed content management system (CMS) adapter.

aclId attribute

Specifies the adapter ID associated with this `CMSAdapter` object. You can use this ID with the Arbortext Command Language (ACL) programming language such as with the `sess_connect()` function. However, such usage is discouraged because the appropriate AOM method should be used instead.

`aclId`	
Access	read-only
Returns	`int`

name attribute

Specifies the human-readable name for this adapter.

`name`	
Access	read-only
Returns	`String`

qualifiedName attribute

Specifies the adapter qualified name associated with this adapter. Each adapter is guaranteed to have a unique qualified name.

`qualifiedName`	
Access	read-only
Returns	`String`

valid attribute

Indicates whether this adapter object is still valid. Some (older) adapters can get unloaded before application exit.

`valid`	
Access	read-only
Returns	`boolean`

connect method

Establishes a content management system (CMS) session. On success, this session will become the "active" session. See the `activeSession` attribute of the `Application` interface for more details.

connect(loginId, password, dmsId)	
Parameters	String *loginId* Specifies the CMS user name. String *password* Specifies the password for the loginId parameter. String *dmsId* Specifies the CMS-specific identifier for the repository domain, library, docbase, and so forth.
Returns	CMSSession. A new CMS session.
Throws	CMSException Raised for any error.

createEvent method

Creates a CMS adapter event.

createEvent(eventType)	
Parameters	String *eventType* Specifies the type of Event interface to be created. The event modules supported by this method are CMSAdapterConnectEvent and CMSAdapterDisconnectEvent. If the Event is to be dispatched with the dispatchEvent method, the appropriate event init method must be called after creation in order to initialize the Event's values. As an example, a user wishing to synthesize a CMSAdapterPreConnect event would call createEvent with the parameter "CMSAdapterPreConnect". The initCMSAdapterConnectEvent method could then be called on the newly created CMSAdapterConnectEvent to set the specific type of CMSAdapterConnectEvent to be dispatched and to set its context information.
Returns	Event. The newly created Event
Throws	AOMException Raised if the implementation does not support the type of Event interface requested.

getUserData method

Retrieves application data from the adapter. This method enables user interface or application code to retrieve named data that was previously stored by calling the `setUserData` method.

`getUserData`(key)	
Parameters	`String` *key* Specifies the unique key used to identify the data.
Returns	`String`. The data associated with the given key, or `null` if there is none.
Throws	CMSException Raised for any error.

hasFeature method

Indicates whether this adapter implements a specified feature.

Note

No feature strings are currently defined.

`hasFeature`(feature)	
Parameters	`String` *feature* Specifies the name of the feature.
Returns	`boolean`. Returns `true` if the feature is supported. Returns `false` if it is not.
Throws	CMSException Raised for any error.

setOldUserData method

Can be used to allow the `connect` method to work with older adapters ("Oracle iFS Adapter" or "Documentum Adapter"). Some older adapters require usage of a "user data" field while connecting.

This stores the given data for use with the **next** call to the `connect` method. After that call, the stored data will be **automatically erased** so it won't affect future calls.

Note

This should only be used with older adapters and will have no effect on newer adapters.

The data is stored directly with this AOM object. If this object is disposed before the method call, the data will not be available for use by the method. To avoid any issues, set the data immediately before making the method call.

`setOldUserData`(data)	
Parameters	`String` *data* Specifies the value to store as the old user data.
Returns	`void`

setUserData method

Stores some application data on the adapter. Any existing data for the same `key` is replaced by the new data. This method enables user interface or application code to associate named data with the adapter, which it can later retrieve by calling the `getUserData` method. User data is not saved between Arbortext Editor or Arbortext Publishing Engine sessions.

Some adapters may support additional arguments to certain methods by having the application call `setUserData` with a predefined key just before calling the method. The adapter documentation will describe any such additional arguments.

`setUserData`(key, data)	
Parameters	`String` *key* Specifies the unique key used to identify the data. `String` *data* Specifies the data to associate with the given key, or `null` to remove any existing data for the key.
Returns	`void`
Throws	CMSException Raised for any error.

38

CMSAdapterConnectEvent interface

The `CMSAdapterConnectEvent` interface provides specific contextual information associated with the `CMSAdapterConnectEvent` extension. These event types notify programmers of events related to logging onto a CMS.

initCMSAdapterConnectEvent method

Initializes the value of an `CMSAdapterConnectEvent` created through the `CMSAdapterConnectEvent` interface. This method should only be called before the `CMSAdapterConnectEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSAdapterConnectEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

39

CMSAdapterDisconnectEvent interface

The `CMSAdapterDisconnectEvent` interface provides specific contextual information associated with the `CMSAdapterDisconnectEvent` extension. These event types notify programmers of events related to logging off a CMS session.

currentUser attribute

Specifies the CMS user name associated with the session.

`currentUser`	
Access	read-only
Returns	`String`

initCMSAdapterDisconnectEvent method

Initializes the value of an `CMSAdapterDisconnectEvent` created through the `CMSAdapterDisconnectEvent` interface. This method should only be called before the `CMSAdapterDisconnectEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSAdapterDisconnectEvent`(typeArg, canBubbleArg, cancelableArg, currentUser)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `String` *currentUser*
Returns	`void`

40

CMSBrowseItem interface

The `CMSBrowseItem` interface contains information returned from `CMSBrowseIterator.getNext()`. `CMSBrowseItem` objects are static and do not reflect changes made after the iterator was created.

CMSItemType enumeration

The `CMSItemType` enumerated type indicates the type of object. Some values may be combined with others, as specified below.

The `CMSItemType` enumeration has the following constants of type `int`.

CMS_ITEM_TYPE_UNKNOWN = 0
This is an unknown object.

CMS_ITEM_TYPE_DOC = 1
This is a document object.

CMS_ITEM_TYPE_FOLDER = 2
This is a folder object.

CMS_ITEM_TYPE_CONTAINER = 4
This constant can be combined with the `CMS_ITEM_TYPE_DOC` constant to indicate that the document object contains other objects.

CMS_ITEM_TYPE_GRAPHIC = 8
This is a graphic or other non-CMS object.

CMS_ITEM_TYPE_DISABLED = 16
This constant can be combined with the `IO_ITEM_TYPE_DOC` and `IO_ITEM_TYPE_GRAPHIC` constants to indicate that the object should be "disabled". This should be used for objects which should not be constructed by the application.

CMSLockStatus enumeration

The `CMSLockStatus` enumerated type indicates the lock status of an object.

The `CMSLockStatus` enumeration has the following constants of type `int`.

CMS_ITEM_LOCKED_UNKNOWN = 0
Cannot determine the lock status.

CMS_ITEM_NOT_LOCKED = 1
The object is not locked or checked out.

CMS_ITEM_NOT_LOCKED_CANT_LOCK = 2
The object is not locked or checked out, but it cannot be locked by the current user.

CMS_ITEM_LOCKED_ME = 3
The object is locked or checked out by the current user.

CMS_ITEM_LOCKED_ME_CANT_EDIT = 4

The object is locked or checked out by the current user, but it cannot be edited. The object may have been checked out in a different context.

CMS_ITEM_LOCKED_OTHER = 5

The object is locked or checked out by another user.

CMS_ITEM_LOCKED_OTHER_CANT_VIEW = 6

The object is locked or checked out by another user, and it cannot be accessed.

applyOverlay attribute

Specifies whether the CMS browser should apply its default icon overlay logic (the corresponding value is 1.) Overlay icons represent an object's lock status. A 0 value indicates that the display icon is a composite icon that includes a lock status icon. This attribute is optional. The default is 1. A 0 value is ignored when no displayIcon is provided.

`applyOverlay`	
Access	read-only
Returns	`unsigned short`
Get throws	CMSException `NOT_IMPLEMENTED_ERR`: Raised if the adapter does not support this feature.

displayIcon attribute

Specifies the graphic icon used to represent this object instance in the Editor's CMS browser. The value is a relative pathname, and the standard search path is used. This attribute is optional and may contain an empty string.

`displayIcon`	
Access	read-only
Returns	`String`
Get throws	CMSException `UNIMPL_ERR`: Raised if the adapter does not support this feature.

fullPath attribute

Specifies the adapter-specific full path name of the CMS object. This attribute is optional and may contain an empty string.

fullPath	
Access	read-only
Returns	String

itemType attribute

Contains a bit mask of the CMSItemType constants.

itemType	
Access	read-only
Returns	unsigned short

lockStatus attribute

Contains one of the CMSLockStatus constants.

lockStatus	
Access	read-only
Returns	unsigned short

logicalId attribute

Specifies the object's Logical ID.

logicalId	
Access	read-only
Returns	String

name attribute

Specifies the human-readable object name.

name	
Access	read-only
Returns	String

revision attribute

Specifies the object's content management system (CMS) version identifier. This attribute is optional and may contain an empty string.

`revision`	
Access	read-only
Returns	`String`

41

CMSBrowseIterator interface

For searching and browsing functions, the adapter returns this iterator over the sequence of results. A `CMSBrowseIterator` object is a static view of the browsing results. It does not reflect any changes made to the content management system (CMS) after the iterator was created.

getNext method

Returns the next item in the sequence.

`getNext`()	
Parameters	None
Returns	`CMSBrowseItem`. The next item.
Throws	CMSException `NOMORE_ERR`: Raised if no more items exist.

hasNext method

Indicates whether there are any more items in the sequence.

`hasNext`()	
Parameters	None
Returns	`boolean`. Returns `true` if the iterator has more items. Returns `false` if it does not.
Throws	CMSException Raised if an error occurs.

42

CMSException exception

Defines the exception thrown by the methods and properties in the Arbortext Object Model (AOM) that work with content management systems (CMS). `CMSException` objects contain an error code, an error message, and an optional detailed message.

The `code` field stores one of the `CMSExceptionCode` constants to indicate the error condition. Arbortext defines the error codes.

The `message` field contains a human-readable description of the error. These messages should be localized.

The `detail` field contains an in-depth description of the error that may be written to a log. The description could be something like a Java stack trace or a detailed error description provided by the CMS. Detailed error descriptions do not need to be localized.

Objects that implement the `CMSException` interface include the following properties:

`unsigned short code`

`String message`

`String detail`

CMSExceptionCode enumeration

An integer indicating the type of CMS error generated.

The `CMSExceptionCode` enumeration has the following constants of type `unsigned short`.

NO_NESTED_TRANS_ERR = 1
Adapter does not support nested transactions.

INVALID_POID_ERR = 2
Invalid POID format.

INVALID_LOGID_ERR = 3
Invalid logical ID format.

INVOKE_FAILED_ERR = 4
Adapter method call failed.

BAD_EXTENSION_ERR = 5
Extension operation does not exist.

NO_LICENSE_ERR = 6
Unable to obtain a license for the adapter.

OBJECT_NOT_FOUND_ERR = 7
Object doesn't exist.

NO_CONFIG_ERR = 8
Error opening configuration file.

UNSUP_PROTO_ERR = 9
This version of the adapter is not supported.

STILL_CONNECTED_ERR = 10
Session still connected.

CANT_CONNECT_ERR = 11
Error connecting.

CANT_DISCONNECT_ERR = 12
Error disconnecting.

CANT_LOGIN_ERR = 13
Invalid login.

CANT_INIT_ERR = 14
Adapter initialization failed.

CANT_ALLOC_ERR = 15
Can't allocate resource.

NO_SGML_INCLUDE_ERR = 16
Xinclude cannot be used with SGML documents.

PARENT_UNLOCKED_ERR = 17
Operation failed because parent is not locked.

OPERATION_CANCELED_ERR = 18
Operation was canceled.

DECLS_LOCKED_ERR = 19
Declarations are locked by another user.

UNSUP_ATTR_ERR = 20
Unsupported attribute.

NOMORE_ERR = 21
No more entries.

UNLOCK_ERR = 22
Object is locked.

PARSE_ERR = 23
Parse error.

RESOURCE_ERR = 24
Out of resource.

FOLDER_ERR = 25
Object is a folder.

READ_ONLY_ERR = 26
Object is read-only.

LEAF_ERR = 27
Object is a leaf.

CONTAINER_ERR = 28
Object is a container.

CANT_CREATE_ERR = 29
Can't create object.

CANT_UNLOCK_ERR = 30
Can't unlock object.

CANT_LOCK_ERR = 31
Can't lock object.

CANT_OPEN_ERR = 32
Can't open object.

BAD_ARG_ERR = 33
Invalid argument.

UNIMP_ERR = 34
Unimplemented operation.

FAIL_ERR = 35
General failure.

NOT_CONTAINER_ERR = 36
Object is not a container.

LOCKED_BY_YOU_ERR = 38
Object is already locked by locker.

LOCKED_BY_OTHER_ERR = 39
Object is locked by another.

TOO_MANY_SESSIONS_ERR = 40
Too many open connections.

TOO_MANY_ADAPTERS_ERR = 41
No more adapters can be registered.

SESS_PREFIX_EXISTS_ERR = 42
Connection already established for this session.

ADAPTER_INIT_FAILED_ERR = 43
Adapter failed to initialize.

CANT_FIND_SYM_ERR = 44
Can't find program symbol.

ADAPTER_LOAD_FAILED_ERR = 45
Adapter failed to load.

INVALID_ATTR_ERR = 46
Attribute value is invalid.

INVOCATION_FAILED_ERR = 47
Error invoking the adapter method.

LOG_ERR = 48
Error opening a log output device.

INVALID_CMSADAPTER_ERR = 49
`CMSAdapter` object is invalid (adapter may have been unloaded).

INVALID_CMSSESSION_ERR = 50
`CMSSession` object is invalid (session may have been disconnected).

INVALID_CMSOBJECT_ERR = 51

`CMSObject` object is invalid (session may have been disconnected or document may have been closed).

ADAPTER_ALREADY_REGISTERED_ERR = 52

An adapter with the same qualified name has already been registered.

OPERATION_NOT_ENABLED_ERR = 53

Method cannot be called in the current state. Some adapters support more than one mode, such as online versus offline editing, and not all operations are allowed in every mode. For example, you might not be able to create new CMS folders while working offline.

43

CMSObject interface

The `CMSObject` interface represents a **reference** to a content management system (CMS) object. If a document references the same child object twice then there will be two different references to that same child CMS object. Each reference will have its own distinct `CMSObject` object that can have different properties from the other. For example, the `start` and `end` properties would be different for each.

CMSSaveFlags enumeration

The `CMSSaveFlags` enumerated type is used to construct the `flags` parameter to the `save` method by ORing any of the following options.

The `CMSSaveFlags` enumeration has the following constants of type `int`.

CMS_SAVE_OBJECT_ATTR = 0x1
Indicates to save the attributes.

Will force the adapter to commit any pending attribute changes for this object into the CMS.

CMS_SAVE_OBJECT_CONTENT = 0x2
Indicates to save the object's content into the CMS.

CMS_SAVE_OBJECT_DECLS = 0x4
Indicates to save the XML/SGML declarations.

For adapters which do not support the separate saving of declarations, just include this along with the `CMS_SAVE_OBJECT_CONTENT` bit since the declarations will be saved with the content.

CMS_SAVE_OBJECT_NO_PI = 0x8
Do not save processing instructions (PIs).

CMS_SAVE_OBJECT_UPDATE_ENT_LINKS = 0x10
Indicates to always update internal references, even if they have not changed.

If set, Arbortext Editor and Arbortext Publishing Engine will call the adapter's `IOObject.modifyChildRefs` method even if the child links have not changed.

CMS_SAVE_OBJECT_CHILDREN = 0x20
Indicates to save the object's children (recursively) when the object is saved.

CMSLockFlags enumeration

The `CMSLockFlags` enumerated type is used to construct the `flags` parameter to the `checkout` method by ORing any of the following options.

The `CMSLockFlags` enumeration has the following constants of type `int`.

CMS_LOCK_FORCE = 0x1
Break existing locks (if supported).

CMSObjectClassType enumeration

The `CMSObjectClassType` enumerated type is used with the `objectClass` read-only attribute.

The `CMSObjectClassType` enumeration has the following constants of type `int`.

CMSOBJECT_CLASS_UNKNOWN = 0
The class type is unknown.

CMSOBJECT_CLASS_CONTAINER = 1
The class type is a virtual document object with children.

CMSOBJECT_CLASS_LEAF = 2
The class type is a virtual document object with no children.

CMSOBJECT_CLASS_EXPANDED_FILE_ENTITY = 3
The class type is an expanded file entity.

CMSOBJECT_CLASS_UNEXPANDED_FILE_ENTITY = 4
The class type is an unexpanded file entity.

CMSOBJECT_CLASS_FILE_ENTITY_WINDOW = 5
The class type is a file entity open for editing in a separate window.

CMSOBJECT_CLASS_INCLUDE = 6
The class type is an included object (via XInclude).

CMSOBJECT_CLASS_FALLBACK = 7
The class type is fallback markup for an XInclude that could not be expanded.

CMSObjectLockStatusType enumeration

The `CMSObjectLockStatusType` enumerated type is used with the `lockStatus` read-only attribute.

The `CMSObjectLockStatusType` enumeration has the following constants of type `int`.

CMSOBJECT_STATUS_UNLOCKED = 1
Indicates that this object is not locked or checked out by any user.

CMSOBJECT_STATUS_LOCKED_BY_ME = 3
Indicates that this object is locked or checked out by the current user.

CMSOBJECT_STATUS_LOCKED_BY_OTHER = 5
Indicates that this object is locked or checked out by another user.

CMSBurstFlags enumeration

The `CMSBurstFlags` enumerated type is used to construct the `flags` parameter to the `burst` method by ORing any of the following options.

The `CMSBurstFlags` enumeration has the following constants of type `int`.

CMS_BURST_SET_METADATA = 0x1

Indicates that, in addition to possibly creating new child objects via bursting, metadata on the object should be set according to the rules in the applicable burst configuration file.

aclId attribute

Specifies the `dobj` ID equivalent to this `CMSObject` object. You can use this ID with the Arbortext Command Language (ACL) programming language. If this object is no longer valid, the attribute value will be `0` (an invalid `dobj` ID).

Each access returns a new `dobj` ID. The caller is responsible for calling the ACL `dobj_close()` function on each returned valid ID. Calling `dobj_close()` does not affect the original `CMSObject` or the IDs returned previously.

`aclId`	
Access	read-only
Returns	`int`

allReferences attribute

Returns a collection of all active object references to the same associated CMS object version.

Each `CMSObject` represents a specific reference (or usage) of a CMS object. If a CMS object references (through File Entity or XInclude) the same child object twice in different parts of the document content, then each reference would have its own `CMSObject` object. Use the `allReferences` attribute to write application code that iterates over all open references to this CMS object. Note that the `class` attribute of each reference may be different.

This attribute can return object references from multiple distinct documents.

`allReferences`	
Access	read-only
Returns	`CMSObjectList`
Get throws	CMSException Raised for any error.

cmsObjectType attribute

Specifies the name of the CMS object type.

`cmsObjectType`	
Access	read-only

Returns	`String`
Get throws	CMSException Raised for any error.

cmsPathName attribute

Specifies the path name of object in the CMS. If the object exists in multiple folders, any one of the folder paths could be returned.

`cmsPathName`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

comment attribute

Specifies the check in or check out comment for the object.

There is currently no standard way of setting the comment for an object. This must be handled in an adapter-specific way.

`comment`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

contentType attribute

Specifies the type of the object's content.

For non-graphics, this may be one of the following values:

- `xml`
- `sgml`
- `html`
- `text`
- `ascii`

For graphics and non-markup documents, this will be a file extension ("bmp", "gif", "jpg", "svg", "doc", etc.).

`contentType`	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

creationDate attribute

Specifies the object's creation date in an adapter-specific human-readable form.

`creationDate`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

enclosingObject attribute

Specifies the object reference that encloses this particular CMS object reference. If this is a top level object, the value is `null`.

For example, if the user inserts reference to a "chapter" object into a checked out "book" object (via File Entity or XInclude) then the `enclosingObject` for the "chapter" object reference would be the containing "book" object.

`enclosingObject`	
Access	read-only
Returns	`CMSObject`
Get throws	CMSException Raised for any error.

encoding attribute

Specifies the character encoding of the object's content.

For most adapters, this attribute would only be available if the object was currently loaded.

`encoding`	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

end attribute

Specifies the last DOM Node associated with the object reference. You can reference a given CMS object in multiple places in either a single document or multiple documents. See `allReferences` for more details.

This may be `null` if this object reference is not currently associated with any DOM Nodes. For example, this could represent a folder object or an object whose content has not yet been loaded into a document.

`end`	
Access	read-only
Returns	`Node`
Get throws	CMSException Raised for any error accessing this attribute.

fullTextIndexed attribute

Indicates whether the document is marked for full text indexing.

`fullTextIndexed`	
Access	read-write
Returns	`boolean`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

hasChildRefs attribute

Specifies whether non-folder objects have any child object references.

A `true` value suggests that the `getChildren` method can safely be called to enumerate the children.

`hasChildRefs`	
Access	read-write
Returns	`boolean`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

instanceDoctypeName attribute

Specifies the object's instance document type name. If the object's content contains a document type declaration such as...

```
<!DOCTYPE book PUBLIC "-//Arbortext//DTD DocBook XML V4.0//EN" "axdocbook.dtd">
```

then this attribute would represent the string `book`. Note that this value has nothing to do with the DTD or Schema associated with this object.

Some XML instances do not contain a document type declaration and so this value would be an empty string.

For most adapters, this attribute would only be available if the object was currently loaded.

`instanceDoctypeName`	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

isFolder attribute

Indicates whether the object is a folder or folder subtype.

isFolder	
Access	read-only
Returns	boolean
Get throws	CMSException Raised for any error.

isLatestVersion attribute

Indicates whether this version is the most recent version of the object on a particular CMS branch.

isLatestVersion	
Access	read-only
Returns	boolean
Get throws	CMSException Raised for any error.

isVirtualDocContainer attribute

Indicates whether the object contains references to child objects which are virtual document objects. Objects that reference all of their children using file entities, XIncludes, and graphic tags are not virtual document containers.

isVirtualDocContainer	
Access	read-write
Returns	boolean
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

lockable attribute

Indicates whether the current user can attempt to lock the object. For example, if another user has the object checked out, this attribute should be `false`.

A `true` value is not a guarantee that a `checkout` will succeed since, for example, another user could have checked this out in the mean time.

lockable	
Access	read-only

Returns	`boolean`
Get throws	CMSException Raised for any error.

lockOwner attribute

Specifies the CMS user name that currently holds the lock. Returns an empty string if the object is not locked.

`lockOwner`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

lockStatus attribute

Specifies the lock status of this CMS object. The value is one of the `CMSObjectLockStatusType` enumerated constants.

`lockStatus`	
Access	read-only
Returns	`int`
Get throws	CMSException Raised for any error.

lockStatusDisplay attribute

Specifies a human-readable string describing the lock status. For example, the value of this attribute could be "locked", "unlocked", and so forth. The returned string can be displayed in the user interface and should be localized.

`lockStatusDisplay`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error accessing this attribute.

logicalId attribute

Specifies the Logical ID of the object used for external binding. A Logical ID identifies a class of objects, any one of which may be selected at any given time. For example, a Logical ID can identify a specific version of a specific CMS object (fixed reference) or the current version (floating reference). Logical IDs are valid across sessions, and are stored inside structured documents. At any time, you can translate a Logical ID into a POID that identifies a specific version of a specific object.

`logicalId`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

modificationDate attribute

Specifies the object's last modification date in an adapter-specific human-readable form.

`modificationDate`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

modified attribute

Will be `true` if the object's content has been modified in memory and has not yet been saved.

`modified`	
Access	read-only
Returns	`boolean`
Get throws	CMSException Raised for any error accessing this attribute.

name attribute

Specifies the name of object. This is normally a human-readable name and is used primarily for display purposes.

name	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

objectClass attribute

Specifies the class of the CMS object. The value is one of the `CMSObjectClassType` enumerated constants.

objectClass	
Access	read-only
Returns	`int`
Get throws	CMSException Raised for any error accessing this attribute.

permission attribute

Specifies the permissions associated with the object in a human-readable string. The format of the string is adapter-specific and is for display purposes only.

permission	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

poid attribute

Specifies the Persistent Object Identifier (POID) associated with the object. This is different from a Logical ID, which can represent different versions of an object over time. For example, the Logical ID could represent the "LATEST" version of the object. The POID always references the same version of the object. An application programmer seldom needs to use a POID. Instead, they should mainly use the `logicalId` attribute.

poid	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error accessing this attribute.

publicId attribute

Specifies the Public ID of the object's DTD or Schema.

For most adapters, this attribute would only be available if the object was currently loaded.

publicId	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

readOnly attribute

Indicates whether the object's content is read-only. Note that this is independent of whether the current user has this object checked out because some adapter's may allow for such a combination.

readOnly	
Access	read-only
Returns	`boolean`
Get throws	CMSException Raised for any error.

session attribute

Specifies the `CMSSession` object associated with this object.

session	
Access	read-only

Returns	`CMSSession`
Get throws	CMSException Raised for any error accessing this attribute.

size attribute

Specifies the size of the object content in bytes. This is optional and some adapters may choose to not implement it.

`size`	
Access	read-only
Returns	`int`
Get throws	CMSException Raised for any error.

start attribute

Specifies the first DOM Node associated with the object reference. You can reference a given CMS object in multiple places in either a single document or multiple documents. See the `allReferences` attribute for more details.

This may be `null` if this object reference is not currently associated with any DOM Nodes. For example, this could represent a folder object or an object whose content has not yet been loaded into a document.

`start`	
Access	read-only
Returns	`Node`
Get throws	CMSException Raised for any error accessing this attribute.

systemId attribute

Specifies the System ID of the object's DTD or Schema.

For most adapters, this attribute would only be available if the object was currently loaded.

`systemId`	
Access	read-write
Returns	`String`

Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

tagName attribute

Specifies the tag name for the top-level element in the object. The value is blank for objects with unstructured content.

`tagName`	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

valid attribute

Indicates whether this still represents a valid object reference. For example, if the associated session has been disconnected then this object reference is considered invalid.

`valid`	
Access	read-only
Returns	`boolean`
Get throws	CMSException Raised for any error.

version attribute

Specifies the CMS version ID of the object in an adapter-specific format. This is for display purposes only.

`version`	
Access	read-only

Returns	`String`
Get throws	CMSException Raised for any error.

burst method

Bursts the checked out object. The bursting process follows the established defaults and specific rules for the associated document type. If the object contains sibling (that is, more than one) top-level elements, it is not burst.

`burst`([flags])	
Parameters	`int` *flags* [optional] Specifies how to burst the object. The value is determined through a bit-wise OR of the `CMSBurstFlags` constants.
Returns	`CMSObject`. This object or, possibly, a new `CMSObject` which has been burst.
Throws	CMSException If an error occurs.

cancelCheckout method

Unlocks the object in the CMS without updating it. The adapter can optionally return the previous version of the object.

`cancelCheckout`( )	
Parameters	None
Returns	`CMSObject`. This object, or possibly a new `CMSObject` representing the previous version.
Throws	CMSException If an error occurs.

checkin method

Checks the object in to the CMS. To properly update the revised object in the CMS, you must save the object before calling this method.

`checkin`()	
Parameters	None

Returns	`CMSObject`. This object, or possibly a new `CMSObject` representing the checked in object.
Throws	CMSException If an error occurs.

checkout method

Locks the CMS object for modification. If the `CMS_LOCK_FORCE` flag is set and the object is locked by another user, the object will be forcibly unlocked if the caller has that right.

The exact semantics of this method are adapter-specific. For example, if a CMS does not support versioning then this may simply "lock" the object to prevent other users from editing it.

`checkout`([flags])	
Parameters	`int` *flags* [optional] Specifies the optional flags controlling the checkout. The value is determined through a bit-wise OR of `CMSLockFlags` constants.
Returns	`CMSObject`. This object, or possibly a new `CMSObject` representing the working copy.
Throws	CMSException Raised for any error.

createEvent method

Creates an event of type `CMSObjectEvent`.

`createEvent`(eventType)	
Parameters	`String` *eventType* Specifies the type of `Event` interface to be created. The only event module supported by this method is `CMSObjectEvents`. If the `Event` is to be dispatched with the `dispatchEvent` method, the appropriate event `init` method must be called after creation in order to initialize the `Event`'s values. As an example, a user wishing to synthesize a `CMSObjectPreCheckin` event would call `createEvent` with the parameter "CMSObjectPreCheckin". The `initCMSObjectEvent`

	method could then be called on the newly created `CMSObjectEvent` to set the specific type of `CMSObjectEvent` to be dispatched and to set its context information.
Returns	`Event`. The newly created `Event`.
Throws	CMSException NOT_SUPPORTED_ERR: Raised if the implementation does not support the type of `Event` interface requested.

deleteObject method

Deletes the object from the CMS. All versions of the object will be deleted. After calling this method, you can no longer use this `CMSObject` object.

If the CMS supports referential integrity, this will fail if any of the deleted object versions are referenced as children of other objects.

`deleteObject`( )	
Parameters	None
Returns	`void`
Throws	CMSException If an error occurs.

getAttribute method

Reads the value of an attribute. Attributes are identified by name. If the attribute has more than one value, an index is used to identify which value to return.

To get the values of multiple attributes, use the `getAttributes` method.

`getAttribute`(attribute [, index])	
Parameters	`String` *attribute* Specifies the attribute name. `int` *index* [optional] Specifies the repeating attribute index (zero-based).
Returns	`String`. The attribute value.
Throws	CMSException Raised for any error.

getAttributes method

Gets the values for a list of attributes. Attributes with a single value are stored as `String` entries in the returned `PropertyMap`. Attributes with multiple values are stored as `StringList` entries.

`getAttributes( [attributes] )`	
Parameters	`StringList` *attributes* [optional] Specifies the array of attributes to retrieve. If the value is `null`, all attributes are retrieved. **Note** If an adapter does not support a `null` value, it will throw a `CMSException` with a `code` value of `UNIMP_ERR`.
Returns	`PropertyMap`. A `PropertyMap` containing the requested attribute names and values.
Throws	CMSException Raised for any error.

getChildren method

Retrieves the contents of a folder or the children of a document object.

`getChildren( )`	
Parameters	None
Returns	`CMSBrowseIterator`. The iterator over the object's children. The iterator returns `CMSBrowseItem` objects.
Throws	CMSException Raised for any error.

getParents method

Returns an iterator over the set of documents that reference this object.

`getParents( )`	
Parameters	None

Returns	`CMSBrowseIterator`. The iterator over the objects that reference the specified object. The iterator returns `CMSBrowseItem` objects.
Throws	CMSException Raised for any error.

getUserData method

Retrieves application data from the object. This method enables user interface or application code to retrieve named data that was previously stored by calling the `setUserData` method.

`getUserData`(key)	
Parameters	`String` *key* Specifies the unique key used to identify the data.
Returns	`String`. The data associated with the given key, or `null` if there is none.
Throws	CMSException If an error occurs.

getVersions method

Returns an iterator over all versions of the object.

`getVersions`( )	
Parameters	None
Returns	`CMSBrowseIterator`. An iterator over all versions of the object. The iterator returns `CMSBrowseItem` objects.
Throws	CMSException Raised for any error.

invokeExtension method

Invokes an adapter-specific extension function. Some adapters provide functionality beyond the standard CMS API.

`invokeExtension`(opcode, map)	
Parameters	`int` *opcode* Specifies the adapter-specific value which identifies the extension method to invoke.

	`PropertyMap` *map* Specifies the collection of adapter-specific parameters to the specified extension method.
Returns	`PropertyMap`. A `PropertyMap` populated with adapter-specific content.
Throws	CMSException Raised for any error calling the extension method.

move method

Moves the object to a new folder in the CMS.

`move`(targetFolder)	
Parameters	`CMSObject` *targetFolder* Specifies the target folder object.
Returns	`void`
Throws	CMSException Raised for any error.

releaseReference method

Releases this reference to the underlying repository object. After this call, most methods on this object will throw a `CMSException` with a `code` value of `INVALID_CMSOBJECT_ERR`. However, the `valid` attribute is always safe to access and will return `false` in this case.

`releaseReference`( )	
Parameters	None
Returns	`void`

save method

Saves a CMS object without checking it in (interim save). The object remains checked out.

Some adapters may support the saving of attributes for objects which are not checked out. See the `CMS_SAVE_OBJECT_ATTR` enumerated constant.

`save`(flags)	
Parameters	`int` *flags* Specifies how to save the object. The value is determined

	through a bit-wise OR of the `CMSSaveFlags` constants.
Returns	`void`
Throws	CMSException If an error occurs.

setAttribute method

Sets the value of an attribute. Attributes are identified by name.

To set the values of multiple attributes, use the `setAttributes` method.

`setAttribute`(attribute, value [, index])	
Parameters	`String` *attribute* Specifies the attribute name. `String` *value* Specifies the attribute value. `int` *index* [optional] Specifies the repeating attribute index (zero-based).
Returns	`void`
Throws	CMSException Raised for any error.

setAttributes method

Sets the values for a list of attributes. The calling function passes a `PropertyMap` containing entries for each of the attributes to be set. Attributes with a single value are stored as `String` entries in the `PropertyMap`. Attributes with multiple values are stored as `StringList` entries.

`setAttributes`(attributeValues)	
Parameters	`PropertyMap` *attributeValues* Specifies the `PropertyMap` containing attribute names and values.
Returns	`void`
Throws	CMSException Raised for any error.

setOldUserData method

This method can be used to allow some properties and methods in this interface to work with older adapters ("Oracle iFS Adapter" or "Documentum Adapter"). Some older adapters require usage of a "user data" field with certain ACL functions (such as those starting with `sess_` or `dobj_`). This allows such functionality of older adapters to be accessed via this AOM interface.

This may be used with the following methods:

- `getChildren()`
- `getParents()`
- `getVersions()`
- `save()`
- `checkout()`
- `checkin()`
- `cancelCheckout()`
- `deleteObject()`
- `move()`

This stores the given data for use with the **next** method call which can make use of it. After that method call, the stored data will be **automatically erased** so it won't affect future calls.

Note

This should only be used with older adapters and will have no affect on newer adapters.

The data is stored directly with this AOM object. If this object is disposed before the method call, the data will not be available for use by the method. To avoid any issues, set the data immediately before making the method call.

`setOldUserData`(data)	
Parameters	`String` *data* Specifies the value to store as the old user data.
Returns	`void`

setUserData method

Stores some application data on the object. Any existing data for the same key is replaced by the new data. This method enables user interface or application code to associate named data with the object, that it can later retrieve by calling the `getUserData` method. User data only exists in memory and is not stored persistently.

Some adapters may support additional arguments to certain methods by having the application call `setUserData` with a predefined key just before calling the method. The adapter documentation will describe any such additional arguments.

`setUserData`(key, data)	
Parameters	`String` *key* Specifies the unique key used to identify the data. `String` *data* Specifies the data to associate with the given key, or `null` to remove any existing data for the key.
Returns	`void`
Throws	CMSException If an error occurs.

44

CMSObjectEvent interface

The `CMSObjectEvent` interface provides specific contextual information associated with the `CMSObjectEvent` extension. These event types notify programmers of important CMS object operations.

end attribute

Specifies an event-dependent DOM end `Node` associated with the event.

`end`	
Access	read-only
Returns	`Node`

errorCode attribute

Used when the event handler wants to cancel the operation or throw an error exception. This can hold any defined `CMSExceptionCode` value. To cancel the operation, call `preventDefault()` and store a value of `OPERATION_CANCELED_ERR` into `errorCode`. To cause an error exception, call `preventDefault()`, store any other defined `CMSExceptionCode` value into `errorCode`, and optionally store a message into `errorMessage`.

`errorCode`	
Access	read-write
Returns	`unsigned short`

errorMessage attribute

Used when the event handler wants to throw an error exception and additionally provide a human-readable error message. To do this, call `preventDefault()`, store the appropriate value into `errorCode`, and store a message into `errorMessage`.

`errorMessage`	
Access	read-write
Returns	`String`

flags attribute

Provides an event-dependent bitmask of information.

`flags`	
Access	read-only
Returns	`int`

result attribute

Represents the event-dependent result of an event.

result	
Access	read-write
Returns	CMSObject

start attribute

Specifies an event-dependent DOM start `Node` associated with the event.

start	
Access	read-only
Returns	Node

initCMSObjectEvent method

Initializes the value of an `CMSObjectEvent` created through the `CMSObjectEvent` interface. This method should only be called before the `CMSObjectEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSObjectEvent`(typeArg, canBubbleArg, cancelableArg, endArg, flagsArg, resultArg, startArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `Node` *endArg* Specifies an event-dependent DOM end `Node` associated with the event. `int` *flagsArg* Provides an event-dependent bitmask of information. `CMSObject` *resultArg* Represents the event-dependent result of an event. `Node` *startArg* Specifies an event-dependent DOM start `Node` associated with the event.
Returns	`void`

45

CMSObjectList interface

The `CMSObjectList` interface provides fast, random access to a collection of `CMSObject`s. Do not confuse this with `CMSBrowseIterator` which provides sequential access to `CMSBrowseItem`s and is used when there are possibly high-latency calls being made into the CMS.

length attribute

Specifies the number of items in the collection.

`length`	
Access	read-only
Returns	`unsigned long`

item method

Returns the item in the collection associated with the `index` parameter. If the `index` parameter value is out of range, this method returns `null`.

`item`(index)	
Parameters	`unsigned long` *index* Specifies the index into the collection.
Returns	`CMSObject`. The requested item from the collection.
Throws	CMSException Raised if an error occurs.

releaseReferences method

For each `CMSObject` in this list, releases a reference to the underling repository object. After this call the list `length` is 0.

`releaseReferences`( )	
Parameters	None
Returns	`void`

46

CMSSession interface

The `CMSSession` interface represents a content management system (CMS) session.

CMSBurstBoundaryType enumeration

The `CMSBurstBoundaryType` enumerated type specifies the available types of bursting that can be configured for any given element. It is used with the `getBurstBoundaryType` method.

The `CMSBurstBoundaryType` enumeration has the following constants of type `int`.

CMS_BURST_NO_BOUNDARY = 0
This element is not configured to be burst.

CMS_BURST_FILE_ENTITY = 1
This element is configured to burst as a file entity.

CMS_BURST_VIRTUAL_DOC = 2
This element is configured to burst as a virtual document.

CMS_BURST_XINCLUDE = 3
This element is configured to burst as an xinclude.

CMSBurstPolicy enumeration

The `CMSBurstPolicy` enumerated type specifies when document bursting should occur. It is used with the `burstPolicy` attribute.

The `CMSBurstPolicy` enumeration has the following constants of type `int`.

CMS_BURST_POLICY_NEVER = 0
The adapter does not support bursting.

CMS_BURST_POLICY_ON_CHECKIN = 1
The adapter performs bursting during document check-in.

CMSCreateFlags enumeration

The `CMSCreateFlags` enumerated type is used to construct the `flags` parameter to the `createObjectFromSubtree` and `createNewObject` methods by ORing any of the following options.

The `CMSCreateFlags` enumeration has the following constants of type `int`.

CMS_CREATE_LOCKED = 0x1
The object is initially locked.

CMS_CREATE_VIRTUAL_CONTAINER = 0x2
The object will be a virtual document container. Objects which will reference child objects via File Entity or XInclude need not specify this flag.

CMSOperationEnabledType enumeration

The `CMSOperationEnabledType` enumerated type is used as the return value of the `verifyOperationEnabledInCurrentState` method.

The `CMSOperationEnabledType` enumeration has the following constants of type `unsigned short`.

CMS_OPERATION_ENABLED = 0
Operation is allowed in the current state.

CMS_OPERATION_NOT_ENABLED = 1
Operation is not allowed in the current state. If any methods in the category are called, they will raise a `CMSException` with error code `CMSException.OPERATION_NOT_ENABLED_ERR`.

CMS_OPERATION_NOT_SUPPORTED = 2
Operation is not supported by the adapter. If any methods in the category are called, they will raise a `CMSException` with error code `CMSException.UNIMP_ERR`.

CMS_OPERATION_UNKNOWN = 3
Operation is not recognized by the adapter. This may be returned if a new category was added to Arbortext Editor but the adapter has not been updated. The caller can assume that methods in the category are enabled (they will throw `CMSException.UNIMP_ERR` if not implemented).

CMSSessBurstFlags enumeration

The `CMSSessBurstFlags` enumerated type is used to construct the `flags` parameter to the `burstDocument` method by ORing any of the following options. The negative flags allow the application developer to override the default session bursting rules.

The `CMSSessBurstFlags` enumeration has the following constants of type `int`.

CMS_BURST_FULLTEXT = 0x0001
Enable full text indexing on the top most object.

CMS_BURST_IMPORT_FILEENTS = 0x0002
Import file entities.

CMS_BURST_NO_IMPORT_FILEENTS = 0x0004
Do not import file entities.

CMS_BURST_IMPORT_GRAPHICS = 0x0008
Import graphic files.

CMS_BURST_NO_IMPORT_GRAPHICS = 0x0010
Do not import graphic files.

CMS_BURST_ELEMENTS = 0x0020
Burst on element boundaries.

CMS_BURST_NO_ELEMENTS = 0x0040
Ignore element boundaries.

CMS_BURST_TOP_FILENAME = 0x0080
Use the file name for the topmost object name.

CMS_BURST_NO_TOP_FILENAME = 0x0100
Do not use the file name for the topmost object name.

CMS_BURST_TOP_LOCK = 0x0200
Lock the topmost object for editing.

CMS_BURST_NO_TOP_LOCK = 0x0400
Do not lock the topmost object.

CMS_BURST_USE_LOCATION_RULES = 0x0800
Follow location rules for child objects even if useroverride=on.

CMS_BURST_CREATE_PARTREF_LINKS = 0x01000
Create part reference links

CMS_BURST_NO_CREATE_PARTREF_LINKS = 0x02000
Do not create part reference links

aclId attribute

Represent the session ID associated with the `CMSSession` object. You can use this ID with the Arbortext Command Language (ACL) programming language. If the session is no longer valid, the `aclId` value is an invalid session ID (`-1`).

`aclId`	
Access	read-only
Returns	`int`

adapter attribute

Specifies the `CMSAdapter` object associated with this session.

`adapter`	
Access	read-only

Returns	`CMSAdapter`
Get throws	CMSException Raised for any error accessing this attribute.

burstPolicy attribute

Represents the burst policy of the adapter. The value is one of the `CMSBurstPolicy` enumerated constants.

`burstPolicy`	
Access	read-only
Returns	`int`
Get throws	CMSException Raised for any error.

burstUserOverride attribute

Set to `true` if this session has the user override set to on for bursting-related options such as object names. This setting allows the user to override certain options that would otherwise be completely dictated by the bursting rules. Set to `false` if it is not. This setting prevents the user from overriding the bursting options.

`burstUserOverride`	
Access	read-only
Returns	`boolean`
Get throws	CMSException Raised for any error.

connected attribute

Set to `true` if the session is still connected. Set to `false` if it is not.

`connected`	
Access	read-only
Returns	`boolean`

currentUser attribute

Specifies the current CMS user name. This will normally match the `loginId` parameter to the `CMSAdapter.connect()` method which established this session.

`currentUser`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

defaultFolder attribute

Specifies the Logical ID of the current user's default folder.

`defaultFolder`	
Access	read-write
Returns	`String`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

fullTextSearch attribute

Indicates whether to index new documents for full-text searching. Not all adapters will implement full text searching.

`fullTextSearch`	
Access	read-write
Returns	`boolean`
Get throws	CMSException Raised for any error.
Set throws	CMSException Raised for any error.

objectReuse attribute

Indicates whether the session supports object reuse during bursting by maintaining a Logical ID and filename cache. See the `setFileMappingEntry` for more details.

`objectReuse`	
Access	read-only
Returns	`boolean`
Get throws	CMSException Raised for any error.

sessionToken attribute

Specifies an adapter-specific session identifier that can be used to make calls directly into the CMS vendor API.

This attribute might not be supported by all adapters.

`sessionToken`	
Access	read-only
Returns	`String`
Get throws	CMSException Raised for any error.

burstDocument method

Bursts the specified file system document using this session. If the specified document contains more than one top-level element, it will not be burst.

`burstDocument`(doc [, name [, folderLogicalId [, flags [, logFile]]]])	
Parameters	`Document` *doc* Specifies the document to burst. The document must be a file system object. `String` *name* [optional] Specifies the name to use for the topmost object. If the value of this parameter is `null` or empty, then the bursting rules are consulted. The adapter is allowed to use a variant of this name if, for example, the CMS disallows two objects with the same name in the same folder. The actual name used can be obtained by accessing the `CMSObject` attribute on the

	`ADocument` after a successful burst and then accessing that object's `name` attribute. `String` *folderLogicalId* [optional] Specifies the destination folder for all objects created. If the value of this parameter is `null` or empty, then the bursting rules are consulted. This may be the Logical ID of a folder or it may be in an adapter-specific path format. If your system-wide bursting configuration specification has the `useroverride` setting set to `on`, then the `folder` parameter provides a user-specified destination folder for **all** objects created. If the value of this parameter is `null` or empty, then the bursting rules determine in which folders the new objects are created. If your system-wide bursting configuration specification has the `useroverride` setting set to `off`, then the `folder` parameter is not used and the bursting rules determine in which folders the new objects are created. See the `burstUserOverride` attribute for more details. `int` *flags* [optional] Specifies the flags which control the bursting behavior. The value is a bit-wise OR of the `CMSSessBurstFlags` constants. The negative flag settings override session defaults. `String` *logFile* [optional] Specifies the full path of a local log file to which this method will append diagnostic information related to bursting this document.
Returns	`void`. The given document is now associated with a new top-level object in the CMS. The `CMSObject` attribute of the `ADocument` interface can be used to access information about the new top-level object.
Throws	CMSException Raised for any error bursting the document.

clearBurstConfig method

This is a special method for administrators to use while they are developing the burst configuration files. It clears out all of the burst configuration settings you have loaded and reloads the system-wide settings. The document type-specific configurations are loaded as they are needed – for example, when a document of that document type is burst. This enables new settings to be tested without having to exit Arbortext Editor. This method does not change the folder that Arbortext Editor uses to load burst configuration files.

`clearBurstConfig`( )	
Parameters	None
Returns	`void`
Throws	CMSException Raised for any error.

createEvent method

Creates a `CMSSession` event.

`createEvent`(eventType)	
Parameters	`String` *eventType* Specifies the type of `Event` interface to be created. The only event module supported by this method is `"CMSSessionEvents"`. If the `Event` is to be dispatched with the `dispatchEvent` method, the appropriate event `init` method must be called after creation in order to initialize the `Event`'s values. As an example, a user wishing to synthesize a `CMSSessionConstructEvent` would call `createEvent` with the parameter "CMSSESSIONCONSTRUCTEVENTS". The `initCMSSessionEvent` method could then be called on the newly created `CMSSessionConstructEvent` to set the specific type of `CMSSessionConstructEvent` to be dispatched and to set its context information.
Returns	`Event`. The newly created `Event`.
Throws	CMSException NOT_SUPPORTED_ERR: Raised if the implementation does not support the type of `Event` interface requested.

createFolder method

Creates a new CMS folder object

`createFolder`(name, folderLogicalId [, objType])	
Parameters	`String` *name* Specifies the name of the new CMS folder object. The adapter is allowed to use a variant of this name if, for example, the CMS disallows two objects with the same name in the same folder. The actual name used can be obtained by accessing the `name` attribute of the returned `CMSObject`. `String` *folderLogicalId* Specifies the folder to put the object in. This may be the Logical ID of a folder or it may be in an adapter-specific path format. `String` *objType* [optional] Specifies the CMS object type for the new folder object.
Returns	`CMSObject`. A new object handle.
Throws	CMSException If an error occurs.

createNewObject method

Creates an empty CMS object of the same type as the specified document. If bursting rules are set up, you may use them to supply or override some of the parameter values.

`createNewObject`(name, folderLogicalId, doc [, flags [, objType]])	
Parameters	`String` *name* Specifies the name of the new CMS object. The adapter is allowed to use a variant of this name if, for example, the CMS disallows two objects with the same name in the same folder. The actual name used can be obtained by accessing the `name` attribute of the returned `CMSObject`. `String` *folderLogicalId* Specifies the folder to put the object in. This may be the Logical ID of a folder or it may be in an adapter-specific path format.

	`Document` *doc* Provides context information for the creation of the object. `int` *flags* [optional] Specifies the creation options. The value is determined by a bit-wise OR of the `CMSCreateFlags` constants. `String` *objType* [optional] Specifies the CMS object type for the new object.
Returns	`CMSObject`. A new object handle.
Throws	CMSException If an error occurs.

createObjectFromSubtree method

Creates a new CMS object, assigning content from an in-memory document. If the new object is a folder or an empty document, use `null` values for the `start` and `end` parameters. If bursting rules are setup, you may use them to supply or override some of the parameter values.

After successful completion, the given DOM Nodes will be associated with the new CMS object. However, this does **not** replace the Nodes with a file entity or XInclude reference and so the association will be lost when the containing document is closed unless some additional action is performed.

`createObjectFromSubtree`(name, folderLogicalId, start, end [, flags [, objType]])	
Parameters	`String` *name* Specifies the name of the new CMS object. The adapter is allowed to use a variant of this name if, for example, the CMS disallows two objects with the same name in the same folder. The actual name used can be obtained by accessing the `name` attribute of the returned `CMSObject`. `String` *folderLogicalId* Specifies the folder to put the object in. This may be the Logical ID of a folder or it may be in an adapter-specific path format. `Node` *start* Specifies the DOM `Node` representing the first node to be

	included in the new object. `Node` *end* Specifies the DOM `Node` representing the last node to be included in the new object. This node should be the same as the start node or a subsequent sibling of it. `int` *flags* [optional] Specifies the creation options. The value is determined by a bit-wise OR of the `CMSCreateFlags` constants. `String` *objType* [optional] Specifies the CMS object type for the new object.
Returns	`CMSObject`. A new object handle.
Throws	CMSException If an error occurs.

disconnect method

Closes the CMS session. Only the `connected` attribute can be safely accessed after this method is called.

`disconnect`( )	
Parameters	None
Returns	`void`
Throws	CMSException Raised for any error.

getAttribute method

Reads the value of a session attribute. Attributes are identified by name. The attribute names supported by this method will vary with each adapter.

`getAttribute`(attribute)	
Parameters	`String` *attribute* Specifies the attribute name.
Returns	`String`. The attribute value.
Throws	CMSException Raised for any error.

getBurstBoundaryType method

For the given `node`, determines the burst boundary type according to the bursting rules associated with this session.

`getBurstBoundaryType`(node)	
Parameters	`Node` *node* Specifies the DOM Node to look up in the burst rules.
Returns	`int`. Returns one of the `CMSBurstBoundaryType` enumerated types describing the bursting rule for the given node.
Throws	CMSException If an error occurs.

getDefaultCreateInfo method

Returns the default object creation information that is not specific to any particular document type. This information is defined in the `atidefaults` configuration file.

The information is returned in a `PropertyMap`. The following table shows the supported key string values:

Key	Value Type	Value Description
`IO_CRE_FILE_REFERENCE`	String	Determines whether Insert and Share Object will create an entity or an XInclude. Allowed values are `xinclude` or `entity`.
`IO_CRE_FULL_TEXT_SEARCH`	Number	Determines whether the object is flagged for full text searching. Allowed values are `0` and `1`.
`IO_CRE_LATEST_ID`	String	Specifies the default version label that indicates an object is the "official" current version.
`IO_CRE_LOGICAL_ID`	String	Specifies the default version label that tells the adapter to load the working copy of an

		object. If no working copy exists, the adapter loads the "official" current copy.
`IO_CRE_MAX_LEN`	Number	Specifies the default maximum name length for CMS objects.
`IO_CRE_ROOT_TYPE`	String	Specifies the root object type for Arbortext Editor and Arbortext Publishing Engine objects.
`IO_CRE_TEMP_ID`	String	Specifies the default version label that indicates an object is a working copy.
`IO_CRE_TOP_LOCKED`	Number	Determines whether the topmost CMS object is locked for editing after bursting. Allowed values are `0` and `1`.

`getDefaultCreateInfo`( )	
Parameters	None
Returns	`PropertyMap`. A `PropertyMap` containing the requested information.
Throws	CMSException If an error occurs.

getFile method

Downloads an object from the CMS to a local file and returns the local path name. This method is typically used to retrieve graphic objects.

`getFile`(logicalId [, notation])	
Parameters	`String` *logicalId* Specifies the Logical ID. `String` *notation* [optional] Specifies the graphic file format, if applicable.

Returns	`String`. A local file name. It can be assumed that the adapter is tracking the files it returns and it will manage them appropriately. The application developer must **not** delete this file.
Throws	CMSException If an error occurs.

getFileMappingEntry method

Checks whether a resolved path name already exists in the CMS. You use this method to avoid creating multiple CMS objects from a single source file – for example, by loading multiple entity references to one file. Before calling this method, use the `objectReuse` attribute to determine if this session is managing file mapping entries or not.

`getFileMappingEntry`(pathname)	
Parameters	`String` *pathname* Specifies the resolved entity path name. This should be a normalized form of a local resource path. For example on Windows-based systems, the following paths all represent the same local resource: • `c:\graphics\engine.jpg` • `c:\Graphics\Engine.JPG` • `c:\graphics\..\graphics\engine.jpg` For this reason, the caller should normalize this path in a consistent manner before calling this method.
Returns	`String`. Returns the associated Logical ID, if it is in the CMS. Returns `null` if it is not.
Throws	CMSException If an error occurs.

getGraphicCreateInfo method

Returns the default creation information for a new graphic object. This information is defined in the `atidefaults` configuration file.

The information is returned in a `PropertyMap`. The following table shows the supported key string values:

Key	Value Type	Value Description
IO_CRE_LOCATION	String	Specifies the default location for new graphics. This may be the Logical ID of a folder or it may be in an adapter-specific path format.
IO_CRE_OBJECT_TYPE	String	Specifies the default object type for graphics.
IO_CRE_LABEL	String	Specifies the default version label for graphics.

getGraphicCreateInfo(graphicNode)	
Parameters	Node *graphicNode* Represents a graphic tag. Elements are designated as graphic tags by a document type's DCF (document configuration file) file or by the document's current Styler stylesheet. The stylesheet overrides the DCF file.
Returns	PropertyMap. A PropertyMap containing the requested information.
Throws	CMSException If an error occurs.

getRangeCreateInfo method

Returns the default creation information for a new object, according to the given start and end Nodes. This information can be defined in a configuration file that is specific to the document type associated with the given Nodes. As a fallback, Arbortext Editor and Arbortext Publishing Engine will use the atidefaults configuration file.

The information is returned in a PropertyMap. The following table shows the supported key string values:

Key	Value Type	Value Description
IO_CRE_NAME	String	Specifies the default name for the new object, according to the default naming rules.
IO_CRE_LOCATION	String	Specifies the default location for the new

		object. This may be the Logical ID of a folder or it may be in an adapter-specific path format.
`IO_CRE_OBJECT_TYPE`	String	Specifies the default object type for the new object.
`IO_CRE_LABEL`	String	Specifies the default version label for the new object.

`getRangeCreateInfo`(start, end, isTop)	
Parameters	`Node` *start* Specifies the first node in the range to consider. `Node` *end* Specifies the last node in the range to consider. This node should be the same as the start node or a subsequent sibling of it. `boolean` *isTop* Indicates whether the "topmost is filename" naming rule is being used. If `true` and the associated burst configuration file has `<namerule rule="topmost-is-filename"/>` as the very first `defaultnamecriteria` then the returned `IO_CRE_NAME` value will be derived from the name (if any) of the document containing the start and end Nodes. Otherwise, the naming rules in the associated burst configuration file are used to generate the name.
Returns	`PropertyMap`. A `PropertyMap` containing the requested information.
Throws	CMSException If an error occurs.

getUserData method

Retrieves application data from the session. This method enables user interface or application code to retrieve named data that was previously stored by calling the `setUserData` method.

`getUserData`(key)	
Parameters	`String` *key* Specifies the unique key used to identify the data.
Returns	`String`. Returns the data associated with the given key. Returns `null` if there is none.
Throws	CMSException Raised for any error.

invokeExtension method

Invokes an adapter-specific extension function. Some adapters provide functionality beyond the standard CMS API.

`invokeExtension`(opcode, map)	
Parameters	`int` *opcode* Specifies the adapter-specific value which identifies the extension method to invoke. `PropertyMap` *map* Specifies the collection of adapter-specific parameters to the specified extension method.
Returns	`PropertyMap`. A `PropertyMap` populated with adapter-specific content.
Throws	CMSException Raised for any error calling the extension method.

logicalIdToPoid method

Translates a Logical ID to a Persistent Object Identifier (POID). POIDs are used internally by Arbortext Editor and Arbortext Publishing Engine and are not normally used by an application developer.

`logicalIdToPoid`(logicalId)	
Parameters	`String` *logicalId* Specifies the Logical ID to translate.
Returns	`String`. A POID.
Throws	CMSException If an error occurs.

objectExists method

Indicates whether an object exists in the CMS. The object is identified by a Logical ID. It is sufficient for this method to ensure that the Logical ID or Persistent Object Identifier (POID) format is correct. To verify the object's actual existence in the CMS, and its accessibility, `Application.constructObject` must be used.

`objectExists`(logicalId)	
Parameters	`String` *logicalId* Specifies a Logical ID.
Returns	`boolean`. Returns `true` if the object exists. Returns `false` if it does not.
Throws	CMSException If an error occurs.

poidToLogicalId method

Translates a Persistent Object Identifier (POID) and version to a Logical ID. POIDs are used internally by Arbortext Editor and Arbortext Publishing Engine and are not normally used by an Application Developer.

`poidToLogicalId`(poid [, label])	
Parameters	`String` *poid* Specifies the POID to translate. `String` *label* [optional] Specifies the optional version label. The syntax is adapter-specific.
Returns	`String`. A Logical ID.
Throws	CMSException If an error occurs.

putFile method

Stores a file in the CMS. If the adapter is tracking imported files (see the `objectReuse` attribute), an entry is created in the persistent lookup table associating the filename with the new Logical ID. See the `getFileMappingEntry` method for more details.

`putFile`(filename, objectName, folderLogicalId, notation [, objType])	
Parameters	`String` *filename*

	Specifies the path name to store. Because the adapter may create a file mapping entry, this path should be normalized. See the `getFileMappingEntry` for more details. `String` *objectName* Specifies the name of the CMS object to create. This is the suggested name. The adapter is allowed to use a variant of this name if, for example, the CMS disallows two objects with the same name in the same folder. The actual name used can be obtained by calling the `Application.constructObject` method with the returned Logical ID and accessing that object's `name` attribute. `String` *folderLogicalId* Specifies the location of the new object in the CMS. This may be the Logical ID of a folder or it may be in an adapter-specific path format. `String` *notation* Specifies the graphic file format. This is optional. `String` *objType* [optional] Specifies the CMS object type for new object. If not supplied, the bursting rules are consulted to determine the object type.
Returns	`String`. The Logical ID of the new object.
Throws	CMSException If an error occurs.

refreshObjectStatus method

To improve performance, the internal implementation keeps track of the lock and read-only status of all constructed objects. This method will cause all constructed objects to refresh this information from the adapter. All appropriate views will be updated (if needed) to reflect any change in an object's status.

`refreshObjectStatus( )`	
Parameters	None
Returns	`void`

search method

Searches the CMS for objects that match the specified search criteria. The `criteria` string is created by the search dialog. Its format is adapter-specific.

`search`(criteria)	
Parameters	`String` *criteria* Specifies the adapter-specific string containing search criteria.
Returns	`CMSBrowseIterator`. An iterator over search results. The iterator returns `CMSBrowseItem` objects.
Throws	CMSException If an error occurs.

setAttribute method

Sets the value of a session attribute. The attribute names supported by this method will vary with each adapter.

`setAttribute`(attribute, value)	
Parameters	`String` *attribute* Specifies the attribute name. `String` *value* Specifies the attribute value.
Returns	`void`
Throws	CMSException Raised for any error.

setFileMappingEntry method

Instructs the adapter to persistently store a path name to a Logical ID association. If a mapping already exists for the path name, it will be replaced with the new Logical ID. Use this method in conjunction with the `getFileMappingEntry` method to prevent creating multiple CMS objects based on a single file.

The `getFileMappingEntry` and `setFileMappingEntry` calls are not atomic. During bursting, a new CMS object is created between the `getFileMappingEntry` and `setFileMappingEntry` calls. If multiple processes are performing burst operations, the result might be multiple CMS objects for the same source file. For example, assume process A and process B both call the `getFileMappingEntry` method at the same time and find that an

association does not currently exist. Both processes then create new CMS objects and call the `setFileMappingEntry` method to create the association. The last `setFileMappingEntry` call takes precedence, and its CMS object will be reused by subsequent burst operations. The other CMS object continues to exist and be referenced by its XML document.

There is no standard way to tell the adapter to remove a path name to Logical ID mapping.

`setFileMappingEntry`(pathname, logicalId)	
Parameters	`String` *pathname* Specifies the resolved entity path name. See the `pathname` parameter to the `getFileMappingEntry` method for information about normalization of this path name. `String` *logicalId* Specifies the associated Logical ID in the CMS.
Returns	`void`
Throws	CMSException If an error occurs.

setOldUserData method

Can be used to allow some methods and properties in this interface to work with older adapters ("Oracle iFS Adapter" or "Documentum Adapter"). Some older adapters require usage of a "user data" field with certain ACL functions (such as those starting with `sess_` or `dobj_`). This allows such functionality of older adapters to be accessed via this AOM interface.

This may be used with the following methods...

- `disconnect()`
- `getFile()`
- `putFile()`
- `createObjectFromSubtree()`
- `createNewObject()`
- `search()`

This stores the given data for use with the **next** method call which can make use of it. After that method call, the stored data will be **automatically erased** so it won't affect future calls.

Note

This should only be used with older adapters and will have **no** effect on newer adapters.

The data is stored directly with this AOM object. If this object is disposed before the method call, the data will not be available for use by the method. To avoid any issues, set the data immediately before making the method call.

`setOldUserData`(data)	
Parameters	`String` *data* Specifies the value to store as the old user data.
Returns	`void`

setUserData method

Stores some application data on the session. Any existing data for the same key is replaced by the new data. This method enables user interface or application code to associate named data with the session, which it can retrieve later by calling the `getUserData` method. User data only exists in memory, and is not stored between sessions.

Some adapters may support additional arguments to certain methods by having the application call `setUserData` with a predefined key just before calling the method. The adapter documentation will describe any such additional arguments.

`setUserData`(key, data)	
Parameters	`String` *key* Specifies the unique key used to identify the data. `String` *data* Specifies the data to associate with the given key, or `null` to remove any existing data for the key.
Returns	`void`
Throws	CMSException Raised for any error.

verifyOperationEnabledInCurrentState method

Indicates whether an operation is allowed in the current state. Some adapters support more than one mode, and different operations may be allowed in each mode. Arbortext Editor does not define what modes or states are possible. Instead, it asks the adapter which operations are enabled in the current state.

<table>
<tr><td colspan="2"><code>verifyOperationEnabledInCurrentState</code>(operation)</td></tr>
<tr><td>Parameters</td><td><code>String</code> operation

Specifies the type of operation. The following table lists the valid strings for the <code>operation</code> parameter, along with the corresponding AOM methods.

<table>
<tr><td>operation</td><td>Methods</td></tr>
<tr><td>createObjectMethods</td><td><code>CMSSession.createNewObject</code>, <code>CMSSession.createObjectFromSubtree</code></td></tr>
<tr><td>createFolderMethods</td><td><code>CMSSession.createFolder</code></td></tr>
<tr><td>burstMethods</td><td><code>CMSObject.burst</code>, <code>CMSSession.burstDocument</code></td></tr>
<tr><td>checkinoutMethods</td><td><code>CMSObject.checkin</code>, <code>CMSObject.checkout</code>, <code>CMSObject.cancelCheckout</code></td></tr>
<tr><td>attributeMethods</td><td><code>CMSObject.getAttribute</code>, <code>CMSObject.getAttributes</code>, <code>CMSObject.setAttribute</code>, <code>CMSObject.setAttributes</code></td></tr>
<tr><td>fileContentMethods</td><td><code>CMSSession.getFile</code>, <code>CMSSession.putFile</code></td></tr>
<tr><td>fileMappingMethods</td><td><code>CMSSession.getFileMappingEntry</code>, <code>CMSSession.putFileMapping Entry</code></td></tr>
<tr><td>deleteObjectMethods</td><td><code>CMSObject.deleteOb</code></td></tr>
</table>
</td></tr>
</table>

	<table><tr><td></td><td>ject</td></tr><tr><td>moveObjectMethods</td><td>CMSObject.moveObject</td></tr><tr><td>searchMethods</td><td>CMSObject.search</td></tr><tr><td>childTraversalMethods</td><td>CMSObject.getChildren</td></tr><tr><td>parentTraversalMethods</td><td>CMSObject.getParents</td></tr><tr><td>versionTraversalMethods</td><td>CMSObject.getVersions</td></tr><tr><td>burstConfigMethods</td><td>CMSSession.getBurstUseroverride, CMSSession.getDefaultCreateInfo, CMSSession.getGraphicCreateInfo, CMSSession.getObjectReuse, CMSSession.getRangeCreateInfo</td></tr></table>
Returns	unsigned short. One of the CMSSession.OPERATION_ constants which indicates whether the operation is enabled in the current state.
Throws	CMSException UNIMP_ERR: Raised if the adapter does not implement this method. In that case, the caller may assume that the operation is enabled.

47

CMSSessionBurstDocumentEvent interface

The `CMSSessionBurstDocumentEvent` interface provides specific contextual information associated with the `CMSSessionBurstDocument` extension. These event types notify programmers of events related to document bursting and the resultant objects created in the repository.

canOverride attribute

If true, for the `CMSSessionBurstDocument` event, the event handler can override the values in the topLevelName and folderLogicalId properties.

`canOverride`	
Access	read-only
Returns	`boolean`

document attribute

For the `CMSSessionBurstDocument` event, this is the document that will be burst. For the `CMSSessionPostBurstDocument` event, this is the document that was burst.

`document`	
Access	read-only
Returns	`Document`

errorCode attribute

Used when the event handler wants to cancel the operation or throw an error exception. This can hold any defined `CMSExceptionCode` value. To cancel the operation, call `preventDefault()` and store a value of `OPERATION_CANCELED_ERR` into `errorCode`. To cause an error exception, call `preventDefault()`, store any other defined `CMSExceptionCode` value into `errorCode`, and optionally store a message into `errorMessage`.

`errorCode`	
Access	read-write
Returns	`unsigned short`

errorMessage attribute

Used when the event handler wants to throw an error exception and additionally provide a human-readable error message. To do this, call `preventDefault()`, store the appropriate value into `errorCode`, and store a message into `errorMessage`.

`errorMessage`	
Access	read-write
Returns	`String`

flags attribute

Creation options. Same as the `flags` parameter of the `CMSSession.createNewObject`.

`flags`	
Access	read-only
Returns	`int`

folderLogicalId attribute

Parent folder for the CMS object.

`folderLogicalId`	
Access	read-write
Returns	`String`

topLevelName attribute

Name of the top-level object which will result from bursting the document. This may be null or empty which means the name will be auto-generated according to the bursting rules for this adapter. For the `CMSSessionBurstDocument` event, the event handler can override this value if `canOverride` is true.

`topLevelName`	
Access	read-write
Returns	`String`

initCMSSessionBurstDocumentEvent method

Initializes the value of an `CMSSessionBurstDocumentEvent` created through the `CMSSessionBurstDocumentEvent` interface. This method should only be called before the `CMSSessionBurstDocumentEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSSessionBurstDocumentEvent`(typeArg, canBubbleArg, cancelableArg, canOverrideArg, topLevelNameArg, folderLogicalIdArg, documentArg, flagsArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `boolean` *canOverrideArg* If true, then, for the `CMSSessionBurstDocument` event, the event handler can override the values in the topLevelName and folderLogicalId properties. `String` *topLevelNameArg* Name of the repository object. `String` *folderLogicalIdArg* Represents the parent folder for the new object. `Document` *documentArg* Represents a full path to a resource (file or HTTP) accessible from the client. `int` *flagsArg* Represents an adapter-specific format specification.
Returns	`void`

48

CMSSessionConstructEvent interface

The `CMSSessionConstructEvent` interface provides specific contextual information associated with the `CMSSessionConstructEvent` extension. These event types notify programmers of operations that construct in-memory representations of repository objects.

errorCode attribute

Used when the event handler wants to cancel the operation or throw an error exception. This can hold any defined `CMSExceptionCode` value. To cancel the operation, call `preventDefault()` and store a value of `OPERATION_CANCELED_ERR` into `errorCode`. To cause an error exception, call `preventDefault()`, store any other defined `CMSExceptionCode` value into `errorCode`, and optionally store a message into `errorMessage`.

`errorCode`	
Access	read-write
Returns	`unsigned short`

errorMessage attribute

Used when the event handler wants to throw an error exception and additionally provide a human-readable error message. To do this, call `preventDefault()`, store the appropriate value into `errorCode`, and store a message into `errorMessage`.

`errorMessage`	
Access	read-write
Returns	`String`

result attribute

The constructed CMS object.

`result`	
Access	read-write
Returns	`CMSObject`

initCMSSessionConstructEvent method

Initializes the value of an `CMSSessionConstructEvent` created through the `CMSSessionConstructEvent` interface. This method should only be called before the `CMSSessionConstructEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSSessionConstructEvent`(typeArg, canBubbleArg, cancelableArg, resultArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `CMSObject` *resultArg* Represents the constructed CMS object.
Returns	`void`

49

CMSSessionCreateEvent interface

The `CMSSessionCreateEvent` interface provides specific contextual information associated with the `CMSSessionCreateEvent` extension. These event types notify programmers of events related to creating new CMS objects in the repository.

end attribute

DOM end `Node` associated with the event.

`end`	
Access	read-only
Returns	`Node`

errorCode attribute

Used when the event handler wants to cancel the operation or throw an error exception. This can hold any defined `CMSExceptionCode` value. To cancel the operation, call `preventDefault()` and store a value of `OPERATION_CANCELED_ERR` into `errorCode`. To cause an error exception, call `preventDefault()`, store any other defined `CMSExceptionCode` value into `errorCode`, and optionally store a message into `errorMessage`.

`errorCode`	
Access	read-write
Returns	`unsigned short`

errorMessage attribute

Used when the event handler wants to throw an error exception and additionally provide a human-readable error message. To do this, call `preventDefault()`, store the appropriate value into `errorCode`, and store a message into `errorMessage`.

`errorMessage`	
Access	read-write
Returns	`String`

flags attribute

Creation options. Same as the `flags` parameter of the `CMSSession.createNewObject`.

`flags`	
Access	read-only
Returns	`int`

folderLogicalId attribute

Parent folder for the new object.

folderLogicalId	
Access	read-write
Returns	String

name attribute

Name of the object being created.

name	
Access	read-write
Returns	String

objType attribute

Adapter-specific object type string.

objType	
Access	read-write
Returns	String

result attribute

The CMS object created.

result	
Access	read-write
Returns	CMSObject

start attribute

DOM start Node associated with the event.

start	
Access	read-only
Returns	Node

version attribute

The object's version number. The value is represented using CMS-specific syntax.

version	
Access	read-only
Returns	String

initCMSSessionCreateEvent method

Initializes the value of an `CMSSessionCreateEvent` created through the `CMSSessionCreateEvent` interface. This method should only be called before the `CMSSessionCreateEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSSessionCreateEvent`(typeArg, canBubbleArg, cancelableArg, nameArg, objTypeArg, folderLogicalIdArg, flagsArg, startArg, endArg, versionArg, resultArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `String` *nameArg* Represents the name of the object being created. `String` *objTypeArg* Represents an adapter-specific object type string. `String` *folderLogicalIdArg* Represents the parent folder for the new object. `int` *flagsArg* Same as the `flags` parameter of the `CMSSession.createNewObject` method. `Node` *startArg* First DOM Node in the object's content. `Node` *endArg* Last DOM Node in the object's content. `String` *versionArg* The object's version number. The value is represented using CMS-specific syntax. `CMSObject` *resultArg* The created CMS object.
Returns	`void`

50

CMSSessionDisconnectEvent interface

The `CMSSessionDisconnectEvent` interface provides specific contextual information associated with the `CMSSessionDisconnectEvent` extension. These event types notify programmers of events related to logging off a CMS session.

currentUser attribute

Specifies the CMS user name associated with the session.

`currentUser`	
Access	read-only
Returns	`String`

initCMSSessionDisconnectEvent method

Initializes the value of an `CMSSessionDisconnectEvent` created through the `CMSSessionDisconnectEvent` interface. This method should only be called before the `CMSSessionDisconnectEvent` has been dispatched using the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSSessionDisconnectEvent`(typeArg, canBubbleArg, cancelableArg, currentUser)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `String` *currentUser* Currently logged on user's name.
Returns	`void`

51

CMSSessionFileEvent interface

The `CMSSessionFileEvent` interface provides specific contextual information associated with the `CMSSessionFileEvent` extension. These event types notify programmers of events related to managing non-textual document objects in the repository.

errorCode attribute

Used when the event handler wants to cancel the operation or throw an error exception. This can hold any defined `CMSExceptionCode` value. To cancel the operation, call `preventDefault()` and store a value of `OPERATION_CANCELED_ERR` into `errorCode`. To cause an error exception, call `preventDefault()`, store any other defined `CMSExceptionCode` value into `errorCode`, and optionally store a message into `errorMessage`.

`errorCode`	
Access	read-write
Returns	`unsigned short`

errorMessage attribute

Used when the event handler wants to throw an error exception and additionally provide a human-readable error message. To do this, call `preventDefault()`, store the appropriate value into `errorCode`, and store a message into `errorMessage`.

`errorMessage`	
Access	read-write
Returns	`String`

folderLogicalId attribute

Parent folder for the CMS object.

`folderLogicalId`	
Access	read-write
Returns	`String`

localPath attribute

Full path to a resource (file or HTTP) accessible from the client.

`localPath`	
Access	read-write
Returns	`String`

logicalId attribute

LogicalId for the object being accessed.

logicalId	
Access	read-only
Returns	String

notation attribute

An adapter-specific format specification.

notation	
Access	read-only
Returns	String

objectName attribute

Name of a repository object.

objectName	
Access	read-only
Returns	String

result attribute

The logical ID of an object created in the repository.

result	
Access	read-write
Returns	String

initCMSSessionFileEvent method

Initializes the value of an CMSSessionFileEvent created through the CMSSessionFileEvent interface. This method should only be called before the CMSSessionFileEvent has been dispatched using the dispatchEvent method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initCMSSessionFileEvent`(typeArg, canBubbleArg, cancelableArg, logicalIdArg, localPathArg, notationArg,objectNameArg, folderLogicalIdArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `String` *logicalIdArg* Represents the logicalId of the object being accessed. `String` *localPathArg* Represents a full path to a resource (file or HTTP) accessible from the client. `String` *notationArg* Represents an adapter-specific format specification. `String` *objectNameArg* Name of the repository object. `String` *folderLogicalIdArg* Represents the parent folder for the new object.
Returns	`void`

52

W3C Comment interface

The `Comment` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

This interface inherits from `CharacterData` and represents the content of a comment, i.e., all the characters between the starting '<!--' and ending '-->'. Note that this is the definition of a comment in XML, and, in practice, HTML, although some HTML tools may implement the full SGML comment structure.

53

Component interface

The `Component` interface is the base interface for all window components.

ComponentType enumeration

The `ComponentType` enumeration is an integer showing which type of component object this is.

The `ComponentType` enumeration has the following constants of type `unsigned short`.

DIALOG_COMPONENT = 1
The component is a `Dialog` object.

FRAME_COMPONENT = 2
The component is a `Window` object.

MENUBAR_COMPONENT = 3
The component is a `MenuBar` object.

MENUITEM_COMPONENT = 4
The component is a `MenuItem` object.

componentType attribute

A code representing the type of the underlying object, as defined by `ComponentType`.

`componentType`	
Access	read-only
Returns	`unsigned short`

firstChild attribute

The first child of this component. If there is no such component, this returns `null`.

`firstChild`	
Access	read-only
Returns	`Component`

lastChild attribute

The last child of this component. If there is no such component, this returns `null`.

`lastChild`	
Access	read-only
Returns	`Component`

nextSibling attribute

The next sibling of this component. If there is no such component, this returns `null`.

`nextSibling`	
Access	read-only
Returns	`Component`

ownerWindow attribute

The `Window` in which this `component` resides.

`ownerWindow`	
Access	read-only
Returns	`Window`

parentComponent attribute

The parent of this `component`. If a component has just created and not yet added to the tree, or if it has been removed from the tree, this is `null`.

`parentComponent`	
Access	read-only
Returns	`Component`

previousSibling attribute

The previous sibling of this component. If there is no such component, this returns `null`.

`previousSibling`	
Access	read-only
Returns	`Component`

text attribute

The text associated with the component. The values vary according to the component type as follows:

Component Type	text
`Dialog`	title of window
`Frame`	title of window

`MenuBar`	#menubar
`MenuItem`	label of menu item

For menu items, you can specify an access key in the label by placing an ampersand (`&`) before the character to be used as the key. For example, to specify `F` as the access key for "File", you should specify the label as "`&File`". The character that follows the ampersand in a label is also known as the `mnemonic` of the menu item. The label for a menu separator is a dash (`-`).

`text`	
Access	read-write
Returns	`String`
Set throws	WindowException INVALID_MODIFICATION_ERR: Raised if the new text is not valid for the component. NO_MODIFICATION_ALLOWED_ERR: Raised if the text of the component cannot be modified.

appendChild method

Appends the component `newChild` to the end of the list of children. If the `newChild` is already in the tree, it is first removed.

`appendChild`(newChild)	
Parameters	`Component` *newChild* The component to append.
Returns	`Component`. The component being appended.
Throws	WindowException HIERARCHY_REQUEST_ERR: Raised if the component is of a type that does not allow children of the type of the `newChild` component, or if the component to append is one of this component's ancestors. WRONG_WINDOW_ERR: Raised if `newChild` was created from a different window than the one that created this component.

insertBefore method

Inserts the component `newChild` before the existing child component `refChild`. If `refChild` is `null`, insert `newChild` at the end of the list of children.

`insertBefore`(newChild [, refChild])	
Parameters	`Component` *newChild* The component to insert. `Component` *refChild* [optional] The reference component. That is, the component before which the new component must be inserted.
Returns	`Component`. The component being inserted.
Throws	WindowException HIERARCHY_REQUEST_ERR: Raised if the component is of a type that does not allow children of the type of the `newChild` component, or if the component to insert is one of this component's ancestors. WRONG_WINDOW_ERR: Raised if `newChild` was created from a different window than the one that created this component. NOT_FOUND_ERR: Raised if `refChild` is not a child of this component.

isSameComponent method

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

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

`isSameComponent`(other)	
Parameters	`Component` *other* The component to test against.
Returns	`boolean`. Returns true if the components are the same, false otherwise.

removeChild method

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

`removeChild`(oldChild)	
Parameters	`Component` *oldChild* The component to remove.
Returns	`Component`. The component being removed.
Throws	WindowException NOT_FOUND_ERR: Raised if `oldChild` is not a child of this component.

replaceChild method

Replaces the child component `oldChild` with `newChild` in the list of children, and return the `oldChild` component.

`replaceChild`(newChild, oldChild)	
Parameters	`Component` *newChild* The new component to put in the child list. `Component` *oldChild* The component being replaced in the child list.
Returns	`Component`. The component being replaced.
Throws	WindowException HIERARCHY_REQUEST_ERR: Raised if the component is of a type that does not allow children of the type of the `newChild` component, or if the component to put in is one of this component's ancestors. WRONG_WINDOW_ERR: Raised if `newChild` was created from a different window than the one that created this component. NOT_FOUND_ERR: Raised if `oldChild` is not a child of this component.

54

Composer interface

The `Composer` interface represents a composition pipeline defined by a Composer Configuration File (CCF). The CCF is an XML document corresponding to the Arbortext Composer DTD.

getDefaultParameters method

Returns the pipeline's default parameters.

`getDefaultParameters`( )	
Parameters	None
Returns	`PropertyMap`. The pipeline default parameters (as specified in the pipeline configuration) or `null` if the pipeline could not be parsed.

getParamDocumentation method

Returns the documentation `DOMString` for a parameter. The result is the content of the `Documentation` child of the `Parameter` element in the `Interface` section of the CCF.

`getParamDocumentation`(parameter)	
Parameters	`String` *parameter* The parameter name.
Returns	`String`. The requested value or `null` if not in the CCF.

getParamEnumerationValues method

Returns an array of options for an `enumeration` parameter. Each entry in the list is a `DOMString` object. The parameter value, as passed to `runPipeline(Map)`, must match an option. The list is the content of the `Value` children of the `Parameter` element in the `Interface` section of the CCF.

`getParamEnumerationValues`(parameter)	
Parameters	`String` *parameter* The parameter name.
Returns	`StringList`. The `List` of options if the `getParamType` is "`enumeration`", otherwise returns `null`.

getParamLabel method

Returns the label `DOMString` for a parameter. The result is the content of the `Label` child of the `Parameter` element in the `Interface` section of the CCF.

`getParamLabel`(parameter)	
Parameters	`String` *parameter* The parameter name.
Returns	`String`. The requested value or `null` if not in the CCF.

getParamType method

Returns the type string for a parameter. The result is the `type` attribute of the `Parameter` element in the `Interface` section of the CCF.

`getParamType`(parameter)	
Parameters	`String` *parameter* The parameter name.
Returns	`String`. The requested value or `null` if not in the CCF.

isParamRequired method

Returns the required flag for a parameter. A composer cannot be run unless all required parameters have been given values. The value is the `required` attribute of this `Parameter` in the `Interface` section of the CCF.

`isParamRequired`(parameter)	
Parameters	`String` *parameter* The parameter name.
Returns	`trueboolean`. if the named parameter is required. Otherwise, `false`.

runPipeline method

Runs the pipeline defined by the CCF using the given map of parameter values.

`runPipeline`([pipelineParameters])	
Parameters	`PropertyMap` *pipelineParameters* [optional] A map object containing the pipeline parameters. The value of a parameter can only be strings.
Returns	`trueboolean`. if the composition run succeeded. Otherwise, `false`.

55

ControlEvent interface

The `ControlEvent` interface provides specific contextual information associated with `Control` events.

initControlEvent method

Initializes the value of a `ControlEvent` created through the `Window createEvent` method. This method should only be called before the `ControlEvent` has been dispatched with the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initControlEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

56

Dialog interface

The `Dialog` interface extends the `Window` interface. It represents a XUI dialog and has the single attribute `dialogView`.

dialogView attribute

The XUI dialog view of the `dialog`. Through this attribute, application programmers can get the XUI document of the `dialog`.

`dialogView`	
Access	read-only
Returns	`View`

57

W3C Document interface

The `Document` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

doctype attribute

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

`doctype`	
Access	read-only
Returns	`DocumentType`

documentElement attribute

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

`documentElement`	
Access	read-only
Returns	`Element`

documentURI attribute

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

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

`documentURI`	
Access	read-write
Returns	`String`

domConfig attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

The configuration used when `Document.normalizeDocument` is invoked.

`domConfig`	
Access	read-only
Returns	`DOMConfiguration`

implementation attribute

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

`implementation`	
Access	read-only
Returns	`DOMImplementation`

inputEncoding attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying the encoding used for this document at the time of the parsing. This is `null` when it is not known, such as when the `Document` was created in memory.

`inputEncoding`	
Access	read-only
Returns	`String`

strictErrorChecking attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying whether error checking is enforced or not. When set to `false`, the implementation is free to not test every possible error case normally defined on DOM operations, and not raise any `DOMException` on DOM operations or report errors while using `Document.normalizeDocument()`. In case of error, the behavior is undefined. This attribute is `true` by default.

`strictErrorChecking`	
Access	read-write
Returns	`boolean`

xmlEncoding attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying, as part of the XML declaration, the encoding of this document. This is `null` when unspecified or when it is not known, such as when the `Document` was created in memory.

`xmlEncoding`	
Access	read-only
Returns	`String`

xmlStandalone attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying, as part of the XML declaration, whether this document is standalone. This is `false` when unspecified.

Note

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

`xmlStandalone`	
Access	read-write
Returns	`boolean`
Set throws	DOMException NOT_SUPPORTED_ERR: Raised if this document does not support the "XML" feature.

xmlVersion attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying, as part of the XML declaration, the version number of this document. If there is no declaration and if this document supports the "XML" feature, the value is `"1.0"` . If this document does not support the "XML" feature, the value is always `null`. Changing this attribute will affect methods that check for invalid characters in XML names. Application should invoke `Document.normalizeDocument()` in order to check for invalid characters in the `Nodes` that are already part of this `Document`.

DOM applications may use the `DOMImplementation.hasFeature(feature, version)` method with parameter values "XMLVersion" and "1.0" (respectively) to determine if an implementation supports [XML 1.0]. DOM applications may use the same method with parameter values "XMLVersion" and "1.1" (respectively) to determine if an implementation supports [XML 1.1]. In both cases, in order to support XML, an implementation must also support the "XML" feature defined in this specification.

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

`xmlVersion`	
Access	read-write
Returns	`String`
Set throws	DOMException NOT_SUPPORTED_ERR: Raised if the version is set to a value that is not supported by this `Document` or if this document does not support the "XML" feature.

adoptNode method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

ATTRIBUTE_NODE
The `ownerElement` attribute is set to `null` and the `specified` flag is set to `true` on the adopted `Attr`. The descendants of the source `Attr` are recursively adopted.

DOCUMENT_FRAGMENT_NODE
The descendants of the source node are recursively adopted.

DOCUMENT_NODE
`Document` nodes cannot be adopted.

DOCUMENT_TYPE_NODE
`DocumentType` nodes cannot be adopted.

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

ENTITY_NODE
`Entity` nodes cannot be adopted.

ENTITY_REFERENCE_NODE
Only the `EntityReference` node itself is adopted, the descendants are discarded, since the source and destination documents might have defined the entity differently. If the document being imported into provides a definition for this entity name, its value is assigned.

NOTATION_NODE
`Notation` nodes cannot be adopted.

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

Note

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

`adoptNode`(source)	
Parameters	`Node` *source* The node to move into this document.
Returns	`Node`. The adopted node, or `null` if this operation fails, such as when the source node comes from a different implementation.
Throws	DOMException NOT_SUPPORTED_ERR: Raised if the source node is of type `DOCUMENT`, `DOCUMENT_TYPE`. NO_MODIFICATION_ALLOWED_ERR: Raised when the source node is readonly.

createAttribute method

Creates an `Attr` of the given name. Note that the `Attr` instance can then be set on an `Element` using the `setAttributeNode` method.

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

`createAttribute`(name)	
Parameters	`String` *name* The name of the attribute.
Returns	`Attr`. A new `Attr` object with the `nodeName` attribute set to `name`, and `localName`, `prefix`, and `namespaceURI` set to `null`. The value of the attribute is the empty string.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

createAttributeNS method

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

`createAttributeNS`(namespaceURI, qualifiedName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to create. `String` *qualifiedName* The qualified name of the attribute to instantiate.

<table>
<tr><td>Returns</td><td>Attr. A new Attr object with the following attributes:
<table>
<tr><td>Node.nodeName</td><td>qualifiedName</td></tr>
<tr><td>Node.namespaceURI</td><td>namespaceURI</td></tr>
<tr><td>Node.prefix</td><td>prefix, extracted from qualifiedName, or null if there is no prefix</td></tr>
<tr><td>Node.localName</td><td>local name, extracted from qualifiedName</td></tr>
<tr><td>Attr.name</td><td>qualifiedName</td></tr>
<tr><td>Node.nodeValue</td><td>the empty string</td></tr>
<tr><td></td><td></td></tr>
</table></td></tr>
<tr><td>Throws</td><td>DOMException
INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character.
NAMESPACE_ERR: Raised if the qualifiedName is malformed, if the qualifiedName has a prefix and the namespaceURI is null, if the qualifiedName has a prefix that is "xml" and the namespaceURI is different from " http://www.w3.org/XML/1998/namespace", or if the qualifiedName is "xmlns" and the namespaceURI is different from " http://www.w3.org/2000/xmlns/".</td></tr>
</table>

createCDATASection method

Creates a CDATASection node whose value is the specified string.

<table>
<tr><td colspan="2">createCDATASection(data)</td></tr>
<tr><td>Parameters</td><td>String data
The data for the CDATASection contents.</td></tr>
<tr><td>Returns</td><td>CDATASection. The new CDATASection object.</td></tr>
<tr><td>Throws</td><td>DOMException
NOT_SUPPORTED_ERR: Raised if this document is an HTML document.</td></tr>
</table>

createComment method

Creates a Comment node given the specified string.

createComment(data)	
Parameters	String *data* The data for the node.
Returns	Comment. The new Comment object.

createDocumentFragment method

Creates an empty DocumentFragment object.

createDocumentFragment( )	
Parameters	None
Returns	DocumentFragment. A new DocumentFragment.

createElement method

Creates an element of the type specified. Note that the instance returned implements the Element interface, so attributes can be specified directly on the returned object.

In addition, if there are known attributes with default values, Attr nodes representing them are automatically created and attached to the element.

To create an element with a qualified name and namespace URI, use the createElementNS method.

createElement(tagName)	
Parameters	String *tagName* The name of the element type to instantiate. For XML, this is case-sensitive. For HTML, the tagName parameter may be provided in any case, but it must be mapped to the canonical uppercase form by the DOM implementation.
Returns	Element. A new Element object with the nodeName attribute set to tagName, and localName, prefix, and namespaceURI set to null.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character.

createElementNS method

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

<table>
<tr><td colspan="2">createElementNS(namespaceURI, qualifiedName)</td></tr>
<tr><td>Parameters</td><td>String namespaceURI
The namespace URI of the element to create.
String qualifiedName
The qualified name of the element type to instantiate.</td></tr>
<tr><td>Returns</td><td>Element. A new Element object with the following attributes:
<table>
<tr><td>Node.nodeName</td><td>qualifiedName</td></tr>
<tr><td>Node.namespaceURI</td><td>namespaceURI</td></tr>
<tr><td>Node.prefix</td><td>prefix, extracted from qualifiedName, or null if there is no prefix</td></tr>
<tr><td>Node.localName</td><td>local name, extracted from qualifiedName</td></tr>
<tr><td>Element.tagName</td><td>qualifiedName</td></tr>
</table></td></tr>
<tr><td>Throws</td><td>DOMException
INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character.
NAMESPACE_ERR: Raised if the qualifiedName is malformed, if the qualifiedName has a prefix and the namespaceURI is null, or if the qualifiedName has a prefix that is "xml" and the namespaceURI is different from " http://www.w3.org/XML/1998/ namespace" [XML Namespaces].</td></tr>
</table>

createEntityReference method

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

Note

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

`createEntityReference`(name)	
Parameters	`String` *name* The name of the entity to reference.
Returns	`EntityReference`. The new `EntityReference` object.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

createProcessingInstruction method

Creates a `ProcessingInstruction` node given the specified name and data strings.

`createProcessingInstruction`(target, data)	
Parameters	`String` *target* The target part of the processing instruction. `String` *data* The data for the node.
Returns	`ProcessingInstruction`. The new `ProcessingInstruction` object.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified target contains an illegal character. NOT_SUPPORTED_ERR: Raised if this document is an HTML document.

createTextNode method

Creates a `Text` node given the specified string.

`createTextNode`(data)	
Parameters	`String` *data* The data for the node.
Returns	`Text`. The new `Text` object.

getElementById method

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

Note

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

`getElementById`(elementId)	
Parameters	`String` *elementId* The unique `id` value for an element.
Returns	`Element`. The matching element.

getElementsByTagName method

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

`getElementsByTagName`(tagName)	
Parameters	`String` *tagName* The name of the tag to match on. The special value "*" matches all tags.
Returns	`NodeList`. A new `NodeList` object containing all the matched `Elements`.

getElementsByTagNameNS method

Returns a `NodeList` of all the `Elements` with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of the `Document` tree.

`getElementsByTagNameNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the elements to match on. The special value "*" matches all namespaces. `String` *localName* The local name of the elements to match on. The special value "*" matches all local names.
Returns	`NodeList`. A new `NodeList` object containing all the matched `Elements`.

importNode method

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

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

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

ATTRIBUTE_NODE

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

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

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

DOCUMENT_NODE
`Document` nodes cannot be imported.

DOCUMENT_TYPE_NODE
`DocumentType` nodes cannot be imported.

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

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

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

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

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

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

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

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

TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE
These three types of nodes inheriting from `CharacterData` copy their `data` and `length` attributes from those of the source node.

`importNode`(importedNode, deep)	
Parameters	`Node` *importedNode* The node to import. `boolean` *deep* If `true`, recursively import the subtree under the specified node; if `false`, import only the node itself, as explained above. This has no effect on `Attr`, `EntityReference`, and `Notation` nodes.
Returns	`Node`. The imported node that belongs to this `Document`.
Throws	DOMException NOT_SUPPORTED_ERR: Raised if the type of node being imported is not supported.

normalizeDocument method

Note
This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

This method acts as if the document was going through a save and load cycle, putting the document in a "normal" form. As a consequence, this method updates the replacement tree of `EntityReference` nodes and normalizes `Text` nodes, as defined in the method `Node.normalize()`.

Otherwise, the actual result depends on the features being set on the `Document.domConfig` object and governing what operations actually take place. Noticeably this method could also make the document namespace well-formed according to the algorithm described in , check the character normalization, remove the `CDATASection` nodes, etc. See `DOMConfiguration` for details.

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

Mutation events, when supported, are generated to reflect the changes occurring on the document.

If errors occur during the invocation of this method, such as an attempt to update a read-only node or a `Node.nodeName` contains an invalid character according to the XML version in use, errors or warnings (`DOMError.SEVERITY_ERROR` or `DOMError.SEVERITY_WARNING`) will be reported using the `DOMErrorHandler` object associated with the " error-handler" parameter. Note this method might also report fatal errors (`DOMError.SEVERITY_FATAL_ERROR`) if an implementation cannot recover from an error.

`normalizeDocument( )`	
Parameters	None
Returns	`void`

renameNode method

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

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

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

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

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

In addition,

- a user data event `NODE_RENAMED` is fired,
- when the implementation supports the feature "MutationNameEvents", each mutation operation involved in this method fires the appropriate event, and in

the end the event { `http://www.w3.org/2001/xml-events`, `DOMElementNameChanged`} or { `http://www.w3.org/2001/xml-events`, `DOMAttributeNameChanged`} is fired.

`renameNode`(n, namespaceURI, qualifiedName)	
Parameters	`Node` *n* The node to rename. `String` *namespaceURI* The new namespace URI. `String` *qualifiedName* The new qualified name.
Returns	`Node`. The renamed node. This is either the specified node or the new node that was created to replace the specified node.
Throws	DOMException NOT_SUPPORTED_ERR: Raised when the type of the specified node is neither `ELEMENT_NODE` nor `ATTRIBUTE_NODE`, or if the implementation does not support the renaming of the document element. INVALID_CHARACTER_ERR: Raised if the new qualified name is not an XML name according to the XML version in use specified in the `Document.xmlVersion` attribute. WRONG_DOCUMENT_ERR: Raised when the specified node was created from a different document than this document. NAMESPACE_ERR: Raised if the `qualifiedName` is a malformed qualified name, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from "http://www.w3.org/XML/1998/namespace" [XML Namespaces]. Also raised, when the node being renamed is an attribute, if the `qualifiedName`, or its prefix, is "xmlns" and the `namespaceURI` is different from " http://www.w3.org/2000/xmlns/".

58

W3C DocumentEditVAL interface

The `DocumentEditVAL` interface is defined in the W3C Document Object Model (DOM) Level 3 Validation Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Val.)

This interface extends the `NodeEditVAL` interface with additional methods for document editing. An object implementing this interface must also implement the `Document` interface.

continuousValidityChecking attribute

An attribute specifying whether the validity of the document is continuously enforced. When the attribute is set to `true`, the implementation may raise certain exceptions, depending on the situation (see the following). This attribute is `false` by default.

`continuousValidityChecking`	
Access	read-write
Returns	`boolean`
Set throws	DOMException NOT_SUPPORTED_ERR: Raised if the implementation does not support setting this attribute to `true`. VALIDATION_ERR: Raised if an operation makes this document not compliant with the `VAL_INCOMPLETE` validity type or the document is invalid, and this attribute is set to `true`. ExceptionVAL NO_SCHEMA_AVAILABLE_ERR: Raised if this attribute is set to `true` and a schema is unavailable.

getDefinedElements method

Returns list of all element information item names of global declaration, belonging to the specified namespace.

`getDefinedElements`(namespaceURI)	
Parameters	`String` *namespaceURI* `namespaceURI` of namespace. For DTDs, this is `null`.
Returns	`NameList`. List of all element information item names belonging to the specified namespace or `null` if no schema is available.

validateDocument method

Validates the document against the schema, e.g., a DTD or an W3C XML schema or another. Any attempt to modify any part of the document while validating results in implementation-dependent behavior. In addition, the validation operation itself cannot modify the document, e.g., for default attributes. This method makes use of the error handler, as described in the [DOM Level 3 Core] `DOMConfiguration` interface, with all errors being `SEVERITY_ERROR` as defined in the `DOMError` interface.

`validateDocument( )`	
Parameters	None
Returns	`unsigned short`. A validation state constant.

59

W3C DocumentEvent interface

The `DocumentEvent` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `DocumentEvent` interface provides a mechanism by which the user can create an Event of a type supported by the implementation. It is expected that the `DocumentEvent` interface will be implemented on the same object which implements the `Document` interface in an implementation which supports the Event model.

createEvent method

createEvent(eventType)	
Parameters	String *eventType* The eventType parameter specifies the type of Event interface to be created. If the Event interface specified is supported by the implementation this method will return a new Event of the interface type requested. If the Event is to be dispatched via the dispatchEvent method the appropriate event init method must be called after creation in order to initialize the Event's values. As an example, a user wishing to synthesize some kind of UIEvent would call createEvent with the parameter "UIEvents". The initUIEvent method could then be called on the newly created UIEvent to set the specific type of UIEvent to be dispatched and set its context information. The createEvent method is used in creating Events when it is either inconvenient or unnecessary for the user to create an Event themselves. In cases where the implementation provided Event is insufficient, users may supply their own Event implementations for use with the dispatchEvent method.
Returns	Event. The newly created Event
Throws	DOMException NOT_SUPPORTED_ERR: Raised if the implementation does not support the type of Event interface requested

60

W3C DocumentFragment interface

The `DocumentFragment` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

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

When a `DocumentFragment` is inserted into a `Document` (or indeed any other `Node` that may take children) the children of the `DocumentFragment` and not the `DocumentFragment` itself are inserted into the `Node`. This makes the `DocumentFragment` very useful when the user wishes to create nodes that

are siblings; the `DocumentFragment` acts as the parent of these nodes so that the user can use the standard methods from the `Node` interface, such as `insertBefore` and `appendChild`.

61

W3C DocumentRange interface

The `DocumentRange` interface is defined in the W3C Document Object Model (DOM) Level 2 Traversal and Range Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113.)

createRange method

This interface can be obtained from the object implementing the `Document` interface using binding-specific casting methods.

`createRange( )`	
Parameters	None
Returns	`Range`. The initial state of the Range returned from this method is such that both of its boundary-points are positioned at the beginning of the corresponding Document, before any content. The Range returned can only be used to select content associated with this Document, or with DocumentFragments and Attrs for which this Document is the `ownerDocument`.

62

W3C DocumentType interface

The `DocumentType` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

entities attribute

A `NamedNodeMap` containing the general entities, both external and internal, declared in the DTD. Parameter entities are not contained. Duplicates are discarded. For example in:

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

the interface provides access to `foo` and the first declaration of `bar` but not the second declaration of `bar` or `baz`. Every node in this map also implements the `Entity` interface.

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

`entities`	
Access	read-only
Returns	`NamedNodeMap`

internalSubset attribute

The internal subset as a string.

Note

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

`internalSubset`	
Access	read-only
Returns	`String`

name attribute

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

name	
Access	read-only
Returns	`String`

notations attribute

A `NamedNodeMap` containing the notations declared in the DTD. Duplicates are discarded. Every node in this map also implements the `Notation` interface.

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

notations	
Access	read-only
Returns	`NamedNodeMap`

publicId attribute

The public identifier of the external subset.

publicId	
Access	read-only
Returns	`String`

systemId attribute

The system identifier of the external subset.

systemId	
Access	read-only
Returns	`String`

63

W3C DocumentView interface

The `DocumentView` interface is defined in the W3C Document Object Model (DOM) Level 2 Views Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113.)

The `DocumentView` interface is implemented by `Document` objects in DOM implementations supporting DOM Views. It provides an attribute to retrieve the default view of a document.

defaultView attribute

The default `AbstractView` for this `Document`, or `null` if none available.

`defaultView`	
Access	read-only
Returns	`AbstractView`

64

W3C DOMConfiguration interface

The `DOMConfiguration` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

Note

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

The following list of parameters defined in the DOM:

"canonical-form"

true

[optional]

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

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

Parameters set to `false`: " entities", "normalize-characters", "cdata-sections".

Parameters set to `true`: "namespaces", "namespace-declarations", " well-formed", "element-content-whitespace".

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

false

[required] (default)

Do not canonicalize the document.

"cdata-sections"

true

[required] (default)

Keep `CDATASection` nodes in the document.

false

[required]

Transform `CDATASection` nodes in the document into `Text` nodes. The new `Text` node is then combined with any adjacent `Text` node.

"check-character-normalization"

true
[optional]

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

false
[required] (default)

Do not check if characters are normalized.

"comments"

true
[required] (default)

Keep `Comment` nodes in the document.

false
[required]

Discard `Comment` nodes in the document.

"datatype-normalization"

true
[optional]

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

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

false
[required] (default)

Do not perform schema normalization on the tree.

"element-content-whitespace"

true
[required] (default)

Keep all whitespaces in the document.

false

[optional]

Discard all `Text` nodes that contain whitespaces in element content, as described in [element content whitespace] . The implementation is expected to use the attribute `Text.isElementContentWhitespace` to determine if a `Text` node should be discarded or not.

"entities"

true

[required] (default)

Keep `EntityReference` nodes in the document.

false

[required]

Remove all `EntityReference` nodes from the document, putting the entity expansions directly in their place. `Text` nodes are normalized, as defined in `Node.normalize`. Only unexpanded entity references are kept in the document.

This parameter does not affect `Entity` nodes.

"error-handler"

[required]

Contains a `DOMErrorHandler` object. If an error is encountered in the document, the implementation will call back the `DOMErrorHandler` registered using this parameter. The implementation may provide a default `DOMErrorHandler` object.

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

"namespaces"

true

[required] (default)

Perform the namespace processing as defined in .

false

[optional]

Do not perform the namespace processing.

"namespace-declarations"

This parameter has no effect if the parameter "namespaces" is set to `false`.

true
[required] (default)

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

false
[required]

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

"normalize-characters"

true
[optional]

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

false
[required] (default)

Do not perform character normalization.

"schema-type"
[optional]

Represent a `DOMString` object containing an absolute URI and representing the type of the schema language used to validate a document against. Note that no lexical checking is done on the absolute URI.

If this parameter is not set, a default value may be provided by the implementation, based on the schema languages supported and on the schema language used at load time. If no value is provided, this parameter is `null`.

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

"validate"

true
[optional]

Require the validation against a schema (i.e. XML schema, DTD, any other type or representation of schema) of the document as it is being normalized as defined by [XML 1.0]. If validation errors are found, or no

schema was found, the error handler is notified. Schema-normalized values will not be exposed according to the schema in used unless the parameter " datatype-normalization" is `true`.

This parameter will reevaluate:

- Attribute nodes with `Attr.specified` equals to `false`, as specified in the description of the `Attr` interface;
- The value of the attribute `Text.isElementContentWhitespace` for all `Text` nodes;
- The value of the attribute `Attr.isId` for all `Attr` nodes;
- The attributes `Element.schemaTypeInfo` and `Attr.schemaTypeInfo`.

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

false

[required] (default)

Do not accomplish schema processing, including the internal subset processing. Default attribute values information are kept. Note that validation might still happen if "validate-if-schema" is `true`.

"validate-if-schema"

true

[optional]

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

"validate-if-schema" and "validate" are mutually exclusive, setting one of them to `true` will set the other one to `false`.

false

[required] (default)

No schema processing should be performed if the document has a schema, including internal subset processing. Default attribute values information are kept. Note that validation must still happen if "validate" is `true`.

"well-formed"

true

[required] (default)

Check if all nodes are XML well formed according to the XML version in use in `Document.xmlVersion`:

- check if the attribute `Node.nodeName` contains invalid characters according to its node type and generate a `DOMError` of type `"wf-invalid-character-in-node-name"`, with a `DOMError.SEVERITY_ERROR` severity, if necessary;
- check if the text content inside `Attr`, `Element`, `Comment`, `Text`, `CDATASection` nodes for invalid characters and generate a `DOMError` of type `"wf-invalid-character"`, with a `DOMError.SEVERITY_ERROR` severity, if necessary;
- check if the data inside `ProcessingInstruction` nodes for invalid characters and generate a `DOMError` of type `"wf-invalid-character"`, with a `DOMError.SEVERITY_ERROR` severity, if necessary;

false

[optional]

Do not check for XML well-formedness.

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

canSetParameter method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

`canSetParameter`(name, value)	
Parameters	`String` *name* The name of the parameter to check. `DOMUserData` *value* An object. if `null`, the returned value is `true`.
Returns	`booleantrue` if the parameter could be successfully set to the specified value, or `false` if the parameter is not recognized or the requested value is not supported. This does not change the current value of the parameter itself.

getParameter method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

Return the value of a parameter if known.

`getParameter`(name)	
Parameters	`String` *name* The name of the parameter.

Returns	`DOMUserData`. The current object associated with the specified parameter or `null` if no object has been associated or if the parameter is not supported. "by a DOM application" prevents a DOM implementation to return its default behavior (such as the default "schema-type") if any.
Throws	DOMException NOT_FOUND_ERR: Raised when the parameter name is not recognized.

setParameter method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

Set the value of a parameter.

`setParameter`(name, value)	
Parameters	`String` *name* The name of the parameter to set. `DOMUserData` *value* The new value or `null` if the user wishes to unset the parameter. While the type of the value parameter is defined as `DOMUserData`, the object type must match the type defined by the definition of the parameter. For example, if the parameter is `"error-handler"`, the value must be of type `DOMErrorHandler`. Should we allow implementations to raise exception if the type does not match? INVALID_ACCESS_ERR seems the closest exception code...
Returns	`void`
Throws	DOMException NOT_SUPPORTED_ERR: Raised when the parameter name is recognized but the requested value cannot be set. NOT_FOUND_ERR: Raised when the parameter name is not recognized.

65

W3C DOMException exception

The `DOMException` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

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

Objects that implement the `DOMException` interface include the following property:

`unsigned short code`

ExceptionCode enumeration

An integer indicating the type of error generated.

Note

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

The `ExceptionCode` enumeration has the following constants of type `unsigned short`.

INDEX_SIZE_ERR = 1
If index or size is negative, or greater than the allowed value

DOMSTRING_SIZE_ERR = 2
If the specified range of text does not fit into a DOMString

HIERARCHY_REQUEST_ERR = 3
If any node is inserted somewhere it doesn't belong

WRONG_DOCUMENT_ERR = 4
If a node is used in a different document than the one that created it (that doesn't support it)

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

NO_DATA_ALLOWED_ERR = 6
If data is specified for a node which does not support data

NO_MODIFICATION_ALLOWED_ERR = 7
If an attempt is made to modify an object where modifications are not allowed

NOT_FOUND_ERR = 8
If an attempt is made to reference a node in a context where it does not exist

NOT_SUPPORTED_ERR = 9
If the implementation does not support the requested type of object or operation.

INUSE_ATTRIBUTE_ERR = 10
If an attempt is made to add an attribute that is already in use elsewhere

INVALID_STATE_ERR = 11
If an attempt is made to use an object that is not, or is no longer, usable.

SYNTAX_ERR = 12
If an invalid or illegal string is specified.

INVALID_MODIFICATION_ERR = 13
If an attempt is made to modify the type of the underlying object.

NAMESPACE_ERR = 14
If an attempt is made to create or change an object in a way which is incorrect with regard to namespaces.

INVALID_ACCESS_ERR = 15
If a parameter or an operation is not supported by the underlying object.

VALIDATION_ERR = 16
If a call to a method such as `insertBefore` or `removeChild` would make the `Node` invalid with respect to "partial validity", this exception would be raised and the operation would not be done. This code is used in DOM Validation Specification. Refer to this specification for further information.

TYPE_MISMATCH_ERR = 17
If the type of an object is incompatible with the expected type of the parameter associated to the object.

66

W3C DOMImplementation interface

The `DOMImplementation` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

createDocument method

Creates an XML `Document` object of the specified type with its document element. HTML-only DOM implementations do not need to implement this method.

`createDocument`(namespaceURI, qualifiedName, doctype)	
Parameters	`String` *namespaceURI* The namespace URI of the document element to create. `String` *qualifiedName* The qualified name of the document element to be created. `DocumentType` *doctype* The type of document to be created or `null`. When `doctype` is not `null`, its `Node.ownerDocument` attribute is set to the document being created.
Returns	`Document`. A new `Document` object.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, or if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from " http://www.w3.org/XML/1998/ namespace" [XML Namespaces]. WRONG_DOCUMENT_ERR: Raised if `doctype` has already been used with a different document or was created from a different implementation.

createDocumentType method

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

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

`createDocumentType`(qualifiedName, publicId, systemId)	
Parameters	`String` *qualifiedName* The qualified name of the document type to be created. `String` *publicId*

	The external subset public identifier. `String` *systemId* The external subset system identifier.
Returns	`DocumentType`. A new `DocumentType` node with `Node.ownerDocument` set to `null`.
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed.

getFeature method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

`getFeature`(feature, version)	
Parameters	`String` *feature* The name of the feature requested. Note that any plus sign "+" prepended to the name of the feature will be ignored since it is not significant in the context of this method. `String` *version* This is the version number of the feature to test.
Returns	`DOMObject`. Returns an object which implements the specialized APIs of the specified feature and version, if any, or `null` if there is no object which implements interfaces associated with that feature. If the `DOMObject` returned by this method implements the `DOMImplementation` interface, it must delegate to the primary core `DOMImplementation` and not return results inconsistent with the primary core `DOMImplementation` such as `hasFeature`, `getFeature`, etc.

hasFeature method

Test if the DOM implementation implements a specific feature.

`hasFeature`(feature, version)	
Parameters	`String` *feature* The name of the feature to test (case-insensitive). The values used by DOM features are defined throughout the DOM Level 2 specifications and listed in the section. The name must be an XML name. To avoid possible conflicts, as a convention, names referring to features defined outside the DOM specification should be made unique by reversing the name of the Internet domain name of the person (or the organization that the person belongs to) who defines the feature, component by component, and using this as a prefix. For instance, the W3C SVG Working Group defines the feature "org.w3c.dom.svg". `String` *version* This is the version number of the feature to test. In Level 2, the string can be either "2.0" or "1.0". If the version is not specified, supporting any version of the feature causes the method to return `true`.
Returns	`trueboolean`. if the feature is implemented in the specified version, `false` otherwise.

67

W3C DOMStringList interface

The `DOMStringList` interface is defined in the W3C Document Object Model (DOM) Level 3 Core Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Core.)

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

length attribute

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

`length`	
Access	read-only
Returns	`unsigned long`

contains method

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

`contains`(str)	
Parameters	`String` *str* The string to look for.
Returns	`booleantrue` if the string has been found, `false` otherwise.

item method

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

`item`(index)	
Parameters	`unsigned long` *index* Index into the collection.
Returns	`String`. The `DOMString` at the `index`th position in the `DOMStringList`, or `null` if that is not a valid index.

68

W3C Element interface

The `Element` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

The `Element` interface represents an element in an HTML or XML document. Elements may have attributes associated with them; since the `Element` interface inherits from `Node`, the generic `Node` interface attribute `attributes` may be used to retrieve the set of all attributes for an element. There are methods on the `Element` interface to retrieve either an `Attr` object by name or an attribute value by name. In XML, where an attribute value may contain entity references,

an `Attr` object should be retrieved to examine the possibly fairly complex subtree representing the attribute value. On the other hand, in HTML, where all attributes have simple string values, methods to directly access an attribute value can safely be used as a convenience.

Note

In DOM Level 2, the method `normalize` is inherited from the `Node` interface where it was moved.

schemaTypeInfo attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

The type information associated with this element.

`schemaTypeInfo`	
Access	read-only
Returns	`TypeInfo`

tagName attribute

The name of the element. For example, in:

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

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

`tagName`	
Access	read-only
Returns	`String`

getAttribute method

Retrieves an attribute value by name.

`getAttribute`(name)	
Parameters	`String` *name* The name of the attribute to retrieve.
Returns	`String`. The `Attr` value as a string, or the empty string if that attribute does not have a specified or default value.

getAttributeNS method

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

`getAttributeNS`(namespaceURI , localName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to retrieve. `String` *localName* The local name of the attribute to retrieve.
Returns	`String`. The `Attr` value as a string, or the empty string if that attribute does not have a specified or default value.

getAttributeNode method

Retrieves an attribute node by name.

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

`getAttributeNode`(name)	
Parameters	`String` *name* The name (`nodeName`) of the attribute to retrieve.
Returns	`Attr`. The `Attr` node with the specified name (`nodeName`) or `null` if there is no such attribute.

getAttributeNodeNS method

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

`getAttributeNodeNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to retrieve. `String` *localName* The local name of the attribute to retrieve.
Returns	`Attr`. The `Attr` node with the specified attribute local name and namespace URI or `null` if there is no such attribute.

getElementsByTagName method

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

`getElementsByTagName`(name)	
Parameters	`String` *name* The name of the tag to match on. The special value "*" matches all tags.
Returns	`NodeList`. A list of matching `Element` nodes.

getElementsByTagNameNS method

Returns a `NodeList` of all the descendant `Elements` with a given local name and namespace URI in the order in which they are encountered in a preorder traversal of this `Element` tree.

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

`getElementsByTagNameNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the elements to match on. The special value "*" matches all namespaces. `String` *localName* The local name of the elements to match on. The special value "*" matches all local names.
Returns	`NodeList`. A new `NodeList` object containing all the matched `Elements`.

hasAttribute method

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

`hasAttribute`(name)	
Parameters	`String` *name* The name of the attribute to look for.
Returns	`trueboolean`. if an attribute with the given name is specified on this element or has a default value, `false` otherwise.

hasAttributeNS method

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

`hasAttributeNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to look for. `String` *localName* The local name of the attribute to look for.
Returns	`trueboolean`. if an attribute with the given local name and namespace URI is specified or has a default value on this element, `false` otherwise.

removeAttribute method

Removes an attribute by name. If the removed attribute is known to have a default value, an attribute immediately appears containing the default value as well as the corresponding namespace URI, local name, and prefix when applicable.

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

`removeAttribute`(name)	
Parameters	`String` *name* The name of the attribute to remove.
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

removeAttributeNS method

Removes an attribute by local name and namespace URI. If the removed attribute has a default value it is immediately replaced. The replacing attribute has the same namespace URI and local name, as well as the original prefix.

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

`removeAttributeNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to remove. `String` *localName*

	The local name of the attribute to remove.
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

removeAttributeNode method

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

`removeAttributeNode`(oldAttr)	
Parameters	`Attr` *oldAttr* The `Attr` node to remove from the attribute list.
Returns	`Attr`. The `Attr` node that was removed.
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if `oldAttr` is not an attribute of the element.

setAttribute method

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

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

`setAttribute`(name, value)	
Parameters	`String` *name* The name of the attribute to create or alter. `String` *value* Value to set in string form.

Returns	`void`
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified name contains an illegal character. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

setAttributeNS method

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

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

`setAttributeNS`(namespaceURI, qualifiedName, value)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to create or alter. `String` *qualifiedName* The qualified name of the attribute to create or alter. `String` *value* The value to set in string form.
Returns	`void`
Throws	DOMException INVALID_CHARACTER_ERR: Raised if the specified qualified name contains an illegal character. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NAMESPACE_ERR: Raised if the `qualifiedName` is malformed, if the `qualifiedName` has a prefix and the `namespaceURI` is `null`, if the `qualifiedName` has a prefix that is "xml" and the `namespaceURI` is different from " http://www.w3.org/XML/1998/namespace", or if the `qualifiedName` is "xmlns" and the `namespaceURI` is different from " http://www.w3.org/2000/xmlns/".

setAttributeNode method

Adds a new attribute node. If an attribute with that name (`nodeName`) is already present in the element, it is replaced by the new one.

To add a new attribute node with a qualified name and namespace URI, use the `setAttributeNodeNS` method.

`setAttributeNode`(newAttr)	
Parameters	`Attr` *newAttr* The `Attr` node to add to the attribute list.
Returns	`Attr`. If the `newAttr` attribute replaces an existing attribute, the replaced `Attr` node is returned, otherwise `null` is returned.
Throws	DOMException WRONG_DOCUMENT_ERR: Raised if `newAttr` was created from a different document than the one that created the element. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. INUSE_ATTRIBUTE_ERR: Raised if `newAttr` is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

setAttributeNodeNS method

Adds a new attribute. If an attribute with that local name and that namespace URI is already present in the element, it is replaced by the new one.

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

`setAttributeNodeNS`(newAttr)	
Parameters	`Attr` *newAttr* The `Attr` node to add to the attribute list.

Returns	`Attr`. If the `newAttr` attribute replaces an existing attribute with the same local name and namespace URI, the replaced `Attr` node is returned, otherwise `null` is returned.
Throws	DOMException WRONG_DOCUMENT_ERR: Raised if `newAttr` was created from a different document than the one that created the element. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. INUSE_ATTRIBUTE_ERR: Raised if `newAttr` is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

setIdAttribute method

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

To specify an attribute by local name and namespace URI, use the `setIdAttributeNS` method.

`setIdAttribute`(name, isId)	
Parameters	`String` *name* The name of the attribute. `boolean` *isId* Whether the attribute is a of type ID.
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

setIdAttributeNS method

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

`setIdAttributeNS`(namespaceURI, localName, isId)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute. `String` *localName* The local name of the attribute. `boolean` *isId* Whether the attribute is a of type ID.
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

setIdAttributeNode method

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

`setIdAttributeNode`(idAttr, isId)	
Parameters	`Attr` *idAttr* The attribute node. `boolean` *isId* Whether the attribute is a of type ID.

Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if the specified node is not an attribute of this element.

69

W3C ElementEditVAL interface

The `ElementEditVAL` interface is defined in the W3C Document Object Model (DOM) Level 3 Validation Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Val.)

This interface extends the `NodeEditVAL` interface with additional methods for guided document editing. An object implementing this interface must also implement the `Element` interface.

This interface also has attributes that are a `NameList` of elements or attributes which can appear in the specified context. Some schema languages, i.e., W3C XML schema, define wildcards which provide for validation of attribute and element information items dependent on their namespace names but independent of their local names.

To expose wildcards, the `NameList` returns the values that represent the namespace constraint:

- `{namespaceURI, name}` is `{null, ##any}` if any;
- `{namespaceURI, name}` is `{namespace_a, ##other}` if not and a namespace name (namespace_a);
- `{namespaceURI, name}` is `{null, ##other}` if not and absent;
- Pairs of `{namespaceURI, name}` with values `{a_namespaceURI | null, null}` if a set whose members are either namespace names or absent.

ContentTypeVAL enumeration

An integer indicating the content type of an element.

The `ContentTypeVAL` enumeration has the following constants of type `unsigned short`.

VAL_EMPTY_CONTENTTYPE = 1
: The content model does not allow any content. If the schema is a W3C XML schema, this corresponds to the `empty` content type; and if the schema is a DTD, this corresponds to the `EMPTY` content model.

VAL_ANY_CONTENTTYPE = 2
: The content model contains unordered child information item(s), i.e., element, processing instruction, unexpanded entity reference, character, and comment information items as defined in the XML Information Set. If the schema is a DTD, this corresponds to the `ANY` content model.

VAL_MIXED_CONTENTTYPE = 3
: The content model contains a sequence of ordered element information items optionally interspersed with character data. If the schema is a W3C XML schema, this corresponds to the `mixed` content type.

VAL_ELEMENTS_CONTENTTYPE = 4
: The content model contains a sequence of element information items optionally separated by whitespace. If the schema is a DTD, this is the `element content` content model; and if the schema is a W3C XML schema, this is the `element-only` content type.

VAL_SIMPLE_CONTENTTYPE = 5
: The content model contains character information items. If the schema is a W3C XML schema, then the element has a content type of `VAL_SIMPLE_CONTENTTYPE` if the type of the element is a `simple type definition`, or the type of the element is a `complexType` whose `{content type}` is a `simple type definition`.

allowedAttributes attribute

A `NameList`, as described in [DOM Level 3 Core], of all possible attribute information items or wildcards that can appear as attributes of this element, or `null` if this element has no context or schema. Duplicate pairs of {namespaceURI, name} are eliminated.

`allowedAttributes`	
Access	read-only
Returns	`NameList`

allowedChildren attribute

A `NameList`, as described in [DOM Level 3 Core], of all possible element information items or wildcards that can appear as children of this element, or `null` if this element has no context or schema. Duplicate pairs of {namespaceURI, name} are eliminated.

`allowedChildren`	
Access	read-only
Returns	`NameList`

allowedFirstChildren attribute

A `NameList`, as described in [DOM Level 3 Core], of all possible element information items or wildcards that can appear as a first child of this element, or `null` if this element has no context or schema. Duplicate pairs of {namespaceURI, name} are eliminated.

`allowedFirstChildren`	
Access	read-only
Returns	`NameList`

allowedNextSiblings attribute

A `NameList`, as described in [DOM Level 3 Core], of all element information items or wildcards that can be inserted as a next sibling of this element, or `null` if this element has no context or schema. Duplicate pairs of {namespaceURI, name} are eliminated.

`allowedNextSiblings`	
Access	read-only
Returns	`NameList`

allowedParents attribute

A `NameList`, as described in [DOM Level 3 Core], of all possible element information items that can appear as a parent this element, or `null` if this element has no context or schema.

`allowedParents`	
Access	read-only
Returns	`NameList`

allowedPreviousSiblings attribute

A `NameList`, as described in [DOM Level 3 Core], of all element information items or wildcards that can be inserted as a previous sibling of this element, or `null` if this element has no context or schema.

`allowedPreviousSiblings`	
Access	read-only
Returns	`NameList`

contentType attribute

The content type of an element as defined above.

`contentType`	
Access	read-only
Returns	`unsigned short`

requiredAttributes attribute

A `NameList`, as described in [DOM Level 3 Core], of required attribute information items that must appear on this element, or `null` if this element has no context or schema.

`requiredAttributes`	
Access	read-only
Returns	`NameList`

canRemoveAttribute method

Verifies if an attribute by the given name can be removed.

`canRemoveAttribute`(attrName)	
Parameters	`String` *attrName* Name of attribute.
Returns	`unsigned short`. A validation state constant.

canRemoveAttributeNS method

Verifies if an attribute by the given local name and namespace can be removed.

`canRemoveAttributeNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the attribute to remove. `String` *localName* Local name of the attribute to be removed.
Returns	`unsigned short`. A validation state constant.

canRemoveAttributeNode method

Determines if an attribute node can be removed.

`canRemoveAttributeNode`(attrNode)	
Parameters	`Node` *attrNode* The `Attr` node to remove from the attribute list.
Returns	`unsigned short`. A validation state constant.

canSetAttribute method

Determines if the value for specified attribute can be set.

`canSetAttribute`(attrName, attrval)	
Parameters	`String` *attrName* Name of attribute. `String` *attrval* Value to be assigned to the attribute.
Returns	`unsigned short`. A validation state constant.

canSetAttributeNS method

Determines if the attribute with given namespace and qualified name can be created if not already present in the attribute list of the element. If the attribute with the same qualified name and namespaceURI is already present in the element's attribute list, it tests whether the value of the attribute and its prefix can be set to the new value.

`canSetAttributeNS`(namespaceURI, qualifiedName, value)	
Parameters	`String` *namespaceURI* `namespaceURI` of namespace. `String` *qualifiedName* Qualified name of attribute. `String` *value* Value to be assigned to the attribute.
Returns	`unsigned short`. A validation state constant.

canSetAttributeNode method

Determines if an attribute node can be added.

`canSetAttributeNode`(attrNode)	
Parameters	`Attr` *attrNode* `Node` in which the attribute can possibly be set.
Returns	`unsigned short`. A validation state constant.

canSetTextContent method

Determines if the text content of this node and its descendants can be set to the string passed in.

`canSetTextContent`(possibleTextContent)	
Parameters	`String` *possibleTextContent* Possible text content string.
Returns	`unsigned short`. A validation state constant.

isElementDefined method

Determines if `name` is defined in the schema. This only applies to global declarations. This method is for non-namespace aware schemas.

`isElementDefined`(name)	
Parameters	`String` *name* Name of element.
Returns	`unsigned short`. A validation state constant.

isElementDefinedNS method

Determines if `name` in this namespace is defined in the current context. Thus not only does this apply to global declarations. but depending on the content, this may also apply to local definitions. This method is for namespace aware schemas.

`isElementDefinedNS`(namespaceURI, name)	
Parameters	`String` *namespaceURI* `namespaceURI` of namespace. `String` *name* Name of element.
Returns	`unsigned short`. A validation state constant.

70

W3C Entity interface

The `Entity` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

The `nodeName` attribute that is inherited from `Node` contains the name of the entity.

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

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

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

An `Entity` node does not have any parent.

Note

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

inputEncoding attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying the encoding used for this entity at the time of parsing, when it is an external parsed entity. This is `null` if it an entity from the internal subset or if it is not known.

`inputEncoding`	
Access	read-only
Returns	`String`

notationName attribute

For unparsed entities, the name of the notation for the entity. For parsed entities, this is `null`.

`notationName`	
Access	read-only
Returns	`String`

publicId attribute

The public identifier associated with the entity, if specified. If the public identifier was not specified, this is `null`.

`publicId`	
Access	read-only
Returns	`String`

systemId attribute

The system identifier associated with the entity, if specified. If the system identifier was not specified, this is `null`.

`systemId`	
Access	read-only
Returns	`String`

xmlEncoding attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying, as part of the text declaration, the encoding of this entity, when it is an external parsed entity. This is `null` otherwise.

`xmlEncoding`	
Access	read-only
Returns	`String`

xmlVersion attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

An attribute specifying, as part of the text declaration, the version number of this entity, when it is an external parsed entity. This is `null` otherwise.

`xmlVersion`	
Access	read-only
Returns	`String`

71

W3C EntityReference interface

The `EntityReference` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

As for `Entity` nodes, `EntityReference` nodes and all their descendants are readonly.

72

W3C Event interface

The `Event` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `Event` interface is used to provide contextual information about an event to the handler processing the event. An object which implements the `Event` interface is generally passed as the first parameter to an event handler. More specific context information is passed to event handlers by deriving additional interfaces from `Event` which contain information directly relating to the type of event they accompany. These derived interfaces are also implemented by the object passed to the event listener.

PhaseType enumeration

An integer indicating which phase of event flow is being processed.

The `PhaseType` enumeration has the following constants of type `unsigned short`.

CAPTURING_PHASE = 1
The current event phase is the capturing phase.

AT_TARGET = 2
The event is currently being evaluated at the target `EventTarget`.

BUBBLING_PHASE = 3
The current event phase is the bubbling phase.

bubbles attribute

Used to indicate whether or not an event is a bubbling event. If the event can bubble the value is true, else the value is false.

`bubbles`	
Access	read-only
Returns	`boolean`

cancelable attribute

Used to indicate whether or not an event can have its default action prevented. If the default action can be prevented the value is true, else the value is false.

`cancelable`	
Access	read-only
Returns	`boolean`

currentTarget attribute

Used to indicate the `EventTarget` whose `EventListeners` are currently being processed. This is particularly useful during capturing and bubbling.

`currentTarget`	
Access	read-only
Returns	`EventTarget`

eventPhase attribute

Used to indicate which phase of event flow is currently being evaluated.

`eventPhase`	
Access	read-only
Returns	`PhaseType`

target attribute

Used to indicate the `EventTarget` to which the event was originally dispatched.

`target`	
Access	read-only
Returns	`EventTarget`

timeStamp attribute

Used to specify the time (in milliseconds relative to the epoch) at which the event was created. Due to the fact that some systems may not provide this information the value of `timeStamp` may be not available for all events. When not available, a value of 0 will be returned. Examples of epoch time are the time of the system start or 0:0:0 UTC 1st January 1970.

`timeStamp`	
Access	read-only
Returns	`DOMTimeStamp`

type attribute

The name of the event (case-insensitive). The name must be an XML name.

`type`	
Access	read-only
Returns	`String`

initEvent method

The `initEvent` method is used to initialize the value of an `Event` created through the `DocumentEvent` interface. This method may only be called before the `Event` has been dispatched via the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times the

final invocation takes precedence. If called from a subclass of `Event` interface only the values specified in the `initEvent` method are modified, all other attributes are left unchanged.

`initEvent`(eventTypeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *eventTypeArg* Specifies the event type. This type may be any event type currently defined in this specification or a new event type.. The string must be an XML name. Any new event type must not begin with any upper, lower, or mixed case version of the string "DOM". This prefix is reserved for future DOM event sets. It is also strongly recommended that third parties adding their own events use their own prefix to avoid confusion and lessen the probability of conflicts with other new events. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

preventDefault method

If an event is cancelable, the `preventDefault` method is used to signify that the event is to be canceled, meaning any default action normally taken by the implementation as a result of the event will not occur. If, during any stage of event flow, the `preventDefault` method is called the event is canceled. Any default action associated with the event will not occur. Calling this method for a non-cancelable event has no effect. Once `preventDefault` has been called it will remain in effect throughout the remainder of the event's propagation. This method may be used during any stage of event flow.

`preventDefault`( )	
Parameters	None
Returns	`void`

stopPropagation method

The `stopPropagation` method is used prevent further propagation of an event during event flow. If this method is called by any `EventListener` the event will cease propagating through the tree. The event will complete dispatch to all listeners on the current `EventTarget` before event flow stops. This method may be used during any stage of event flow.

`stopPropagation( )`	
Parameters	None
Returns	`void`

73

W3C EventException exception

The `EventException` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

Event operations may throw an `EventException` as specified in their method descriptions.

Objects that implement the `EventException` interface include the following property:

`unsigned short code`

EventExceptionCode enumeration

An integer indicating the type of error generated.

The `EventExceptionCode` enumeration has the following constants of type `unsigned short`.

UNSPECIFIED_EVENT_TYPE_ERR = 0

If the `Event`'s type was not specified by initializing the event before the method was called. Specification of the Event's type as `null` or an empty string will also trigger this exception.

74

W3C EventListener interface

The `EventListener` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `EventListener` interface is the primary method for handling events. Users implement the `EventListener` interface and register their listener on an `EventTarget` using the `AddEventListener` method. The users should also remove their `EventListener` from its `EventTarget` after they have completed using the listener.

When a `Node` is copied using the `cloneNode` method the `EventListener` s attached to the source `Node` are not attached to the copied `Node`. If the user wishes the same `EventListeners` to be added to the newly created copy the user must add them manually.

handleEvent method

This method is called whenever an event occurs of the type for which the `EventListener` interface was registered.

`handleEvent`(evt)	
Parameters	`Event` *evt* The `Event` contains contextual information about the event. It also contains the `stopPropagation` and `preventDefault` methods which are used in determining the event's flow and default action.
Returns	`void`

75

W3C EventTarget interface

The `EventTarget` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `EventTarget` interface is implemented by all `Nodes` in an implementation which supports the DOM Event Model. Therefore, this interface can be obtained by using binding-specific casting methods on an instance of the `Node` interface. The interface allows registration and removal of `EventListeners` on an `EventTarget` and dispatch of events to that `EventTarget`.

addEventListener method

This method allows the registration of event listeners on the event target. If an `EventListener` is added to an `EventTarget` while it is processing an event, it will not be triggered by the current actions but may be triggered during a later stage of event flow, such as the bubbling phase.

If multiple identical `EventListeners` are registered on the same `EventTarget` with the same parameters the duplicate instances are discarded. They do not cause the `EventListener` to be called twice and since they are discarded they do not need to be removed with the `removeEventListener` method.

`addEventListener`(type, listener, useCapture)	
Parameters	`String` *type* The event type for which the user is registering `EventListener` *listener* The `listener` parameter takes an interface implemented by the user which contains the methods to be called when the event occurs. `boolean` *useCapture* If true, `useCapture` indicates that the user wishes to initiate capture. After initiating capture, all events of the specified type will be dispatched to the registered `EventListener` before being dispatched to any `EventTargets` beneath them in the tree. Events which are bubbling upward through the tree will not trigger an `EventListener` designated to use capture.
Returns	`void`

dispatchEvent method

This method allows the dispatch of events into the implementations event model. Events dispatched in this manner will have the same capturing and bubbling behavior as events dispatched directly by the implementation. The target of the event is the `EventTarget` on which `dispatchEvent` is called.

`dispatchEvent`(evt)	
Parameters	`Event` *evt* Specifies the event type, behavior, and contextual information to be used in processing the event.

Returns	`boolean`. The return value of `dispatchEvent` indicates whether any of the listeners which handled the event called `preventDefault`. If `preventDefault` was called the value is false, else the value is true.
Throws	EventException UNSPECIFIED_EVENT_TYPE_ERR: Raised if the `Event`'s type was not specified by initializing the event before `dispatchEvent` was called. Specification of the `Event`'s type as `null` or an empty string will also trigger this exception.

removeEventListener method

This method allows the removal of event listeners from the event target. If an `EventListener` is removed from an `EventTarget` while it is processing an event, it will not be triggered by the current actions. `EventListeners` can never be invoked after being removed.

Calling `removeEventListener` with arguments which do not identify any currently registered `EventListener` on the `EventTarget` has no effect.

`removeEventListener`(type, listener, useCapture)	
Parameters	`String` *type* Specifies the event type of the `EventListener` being removed. `EventListener` *listener* The `EventListener` parameter indicates the `EventListener` to be removed. `boolean` *useCapture* Specifies whether the `EventListener` being removed was registered as a capturing listener or not. If a listener was registered twice, one with capture and one without, each must be removed separately. Removal of a capturing listener does not affect a non-capturing version of the same listener, and vice versa.
Returns	`void`

76

W3C ExceptionVAL exception

The `ExceptionVAL` interface is defined in the W3C Document Object Model (DOM) Level 3 Validation Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Val.)

Some Validation operations may throw an `ExceptionVAL` as described in their descriptions.

Objects that implement the `ExceptionVAL` interface include the following property:

`unsigned short code`

ExceptionVALCode enumeration

An integer indicating the type of error generated.

The `ExceptionVALCode` enumeration has the following constants of type `unsigned short`.

NO_SCHEMA_AVAILABLE_ERR = 71

This error occurs when the operation cannot complete due to an unavailable schema.

77

MenuBar interface

The `MenuBar` interface represents a menu bar.

find method

Finds the menu item associated with the menu path specified by `menuPath`. A menu path is a way of indicating the exact location of the menu item in the menu bar hierarchy. This item can be either an item on a menu or the menu itself.

Menu path syntax is similar to path name syntax, with periods separating the components instead of slashes. The leading parts of a menu path correspond to a menu name, the trailing part matches a menu item. For example, `.File.New` refers to the item **New** on the **File** menu. Menu paths need not specify the trailing ellipsis on a menu item, for example, `.File.Open` and `.File.Open...` refer to the same menu item.

A menu path is considered absolute if it starts with a period (`.`). The name following the period must be the name of one of the top-level menus on the menu bar (for example, `.File` , `.Edit`, `.Tools`). If a menu path does not start with a period, the entire hierarchy for the menu bar is searched for the first occurrence of the item. The search starts with the first menu on the left and progresses down through every item on a menu before moving on to the next menu to the right.

The syntax of a menu path allows specifications of a menu item by position and also by name. If a component of a menu path begins with `#` and is followed by one or more digits, it specifies a numeric position. For example, the menu path `.File.#3` specifies the third item in the **File** menu. Numeric positions may be specified in any component. For example, `.View.#5.#3` is the same as `.View.Tools.Table` (assuming the default menu configuration). Blank or separator lines within the menu count as items.

Menu item labels may contain ACL variable references. If a menu label contains any variable references (for example, `Modify $tagname`), the variable reference is substituted into the label string each time the menu containing the item is posted.

The find method recognizes the following special characters when matching a menu path against the menu hierarchy:

.	Separates components of menu names. If it is the first character, it is an absolute path to a menu or item within the menu bar.
*	Matches 0 or more characters.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A range of characters can be specified by separating the start and end characters with a hyphen, such as `0-9`, `a-z`, or `A-z` (for all letters, uppercase and lowercase).
_	Matches a space or an underscore.
\	Treat the following special character as an ordinary character. For example, `\.` matches a period.

`find`(menuPath)	
Parameters	`String` *menuPath* The menu path of a menu item.
Returns	`MenuItem`. The `MenuItem` which is associated with the `menuPath`

78

MenuEvent interface

The `MenuEvent` interface provides specific contextual information associated with `Menu` events.

initMenuEvent method

Initializes the value of a `MenuEvent` created through the `Window createEvent` method. This method should only be called before the `MenuEvent` has been dispatched with the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initMenuEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

79

MenuItem interface

The `MenuItem` interface represents a menu item.

checked attribute

For toggle menu items only. Shows whether the toggle menu item is checked or not.

`checked`	
Access	read-write
Returns	`boolean`
Set throws	WindowException INVALID_MODIFICATION_ERR: Raised if the object is not a toggle menu item.

enabled attribute

Shows whether the menu item is active or not.

`enabled`	
Access	read-write
Returns	`boolean`
Set throws	WindowException INVALID_MODIFICATION_ERR: Raised if the object is a top-level menu item which cannot be disabled.

80

W3C MouseEvent interface

The `MouseEvent` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `MouseEvent` interface provides specific contextual information associated with Mouse events.

The `detail` attribute inherited from `UIEvent` indicates the number of times a mouse button has been pressed and released over the same screen location during a user action. The attribute value is 1 when the user begins this action and increments by 1 for each full sequence of pressing and releasing. If the user moves the mouse between the mousedown and mouseup the value will be set to 0, indicating that no click is occurring.

In the case of nested elements mouse events are always targeted at the most deeply nested element. Ancestors of the targeted element may use bubbling to obtain notification of mouse events which occur within its descendent elements.

altKey attribute

Used to indicate whether the 'alt' key was depressed during the firing of the event. On some platforms this key may map to an alternative key name.

`altKey`	
Access	read-only
Returns	`boolean`

button attribute

During mouse events caused by the depression or release of a mouse button, `button` is used to indicate which mouse button changed state. The values for `button` range from zero to indicate the left button of the mouse, one to indicate the middle button if present, and two to indicate the right button. For mice configured for left handed use in which the button actions are reversed the values are instead read from right to left.

`button`	
Access	read-only
Returns	`unsigned short`

clientX attribute

The horizontal coordinate at which the event occurred relative to the DOM implementation's client area.

`clientX`	
Access	read-only
Returns	`long`

clientY attribute

The vertical coordinate at which the event occurred relative to the DOM implementation's client area.

`clientY`	
Access	read-only
Returns	`long`

ctrlKey attribute

Used to indicate whether the 'ctrl' key was depressed during the firing of the event.

ctrlKey	
Access	read-only
Returns	boolean

metaKey attribute

Used to indicate whether the 'meta' key was depressed during the firing of the event. On some platforms this key may map to an alternative key name.

metaKey	
Access	read-only
Returns	boolean

relatedTarget attribute

Used to identify a secondary EventTarget related to a UI event. Currently this attribute is used with the mouseover event to indicate the EventTarget which the pointing device exited and with the mouseout event to indicate the EventTarget which the pointing device entered.

relatedTarget	
Access	read-only
Returns	EventTarget

screenX attribute

The horizontal coordinate at which the event occurred relative to the origin of the screen coordinate system.

screenX	
Access	read-only
Returns	long

screenY attribute

The vertical coordinate at which the event occurred relative to the origin of the screen coordinate system.

screenY	
Access	read-only
Returns	long

shiftKey attribute

Used to indicate whether the 'shift' key was depressed during the firing of the event.

`shiftKey`	
Access	read-only
Returns	`boolean`

initMouseEvent method

The `initMouseEvent` method is used to initialize the value of a `MouseEvent` created through the `DocumentEvent` interface. This method may only be called before the `MouseEvent` has been dispatched via the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initMouseEvent`(typeArg, canBubbleArg, cancelableArg, viewArg, detailArg, screenXArg, screenYArg, clientXArg, clientYArg, ctrlKeyArg, altKeyArg, shiftKeyArg, metaKeyArg, buttonArg, relatedTargetArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `AbstractView` *viewArg* Specifies the `Event`'s `AbstractView`. `long` *detailArg* Specifies the `Event`'s mouse click count. `long` *screenXArg* Specifies the `Event`'s screen x coordinate `long` *screenYArg* Specifies the `Event`'s screen y coordinate `long` *clientXArg* Specifies the `Event`'s client x coordinate `long` *clientYArg* Specifies the `Event`'s client y coordinate `boolean` *ctrlKeyArg* Specifies whether or not control key was depressed during the `Event`. `boolean` *altKeyArg* Specifies whether or not alt key was depressed during the `Event`. `boolean` *shiftKeyArg* Specifies whether or not shift key was depressed during the `Event`. `boolean` *metaKeyArg* Specifies whether or not meta key was depressed during the `Event`. `unsigned short` *buttonArg* Specifies the `Event`'s mouse button. `EventTarget` *relatedTargetArg* Specifies the `Event`'s related `EventTarget`.

Returns	`void`

81

W3C MutationEvent interface

The `MutationEvent` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `MutationEvent` interface provides specific contextual information associated with Mutation events.

AttrChangeType enumeration

An integer indicating in which way the `Attr` was changed.

The `AttrChangeType` enumeration has the following constants of type `unsigned short`.

MODIFICATION = 1
The `Attr` was modified in place.

ADDITION = 2
The `Attr` was just added.

REMOVAL = 3
The `Attr` was just removed.

attrChange attribute

`attrChange` indicates the type of change which triggered the DOMAttrModified event. The values can be `MODIFICATION`, `ADDITION`, or `REMOVAL`.

`attrChange`	
Access	read-only
Returns	`AttrChangeType`

attrName attribute

`attrName` indicates the name of the changed `Attr` node in a DOMAttrModified event.

`attrName`	
Access	read-only
Returns	`String`

newValue attribute

`newValue` indicates the new value of the `Attr` node in DOMAttrModified events, and of the `CharacterData` node in DOMCharDataModified events.

`newValue`	
Access	read-only
Returns	`String`

prevValue attribute

`prevValue` indicates the previous value of the `Attr` node in DOMAttrModified events, and of the `CharacterData` node in DOMCharDataModified events.

`prevValue`	
Access	read-only
Returns	`String`

relatedNode attribute

`relatedNode` is used to identify a secondary node related to a mutation event. For example, if a mutation event is dispatched to a node indicating that its parent has changed, the `relatedNode` is the changed parent. If an event is instead dispatched to a subtree indicating a node was changed within it, the `relatedNode` is the changed node. In the case of the DOMAttrModified event it indicates the `Attr` node which was modified, added, or removed.

`relatedNode`	
Access	read-only
Returns	`Node`

initMutationEvent method

The `initMutationEvent` method is used to initialize the value of a `MutationEvent` created through the `DocumentEvent` interface. This method may only be called before the `MutationEvent` has been dispatched via the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initMutationEvent`(typeArg, canBubbleArg, cancelableArg, relatedNodeArg, prevValueArg, newValueArg, attrNameArg, attrChangeArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `Node` *relatedNodeArg* Specifies the `Event`'s related Node. `String` *prevValueArg* Specifies the `Event`'s `prevValue` attribute. This value may be null. `String` *newValueArg* Specifies the `Event`'s `newValue` attribute. This value may be null. `String` *attrNameArg* Specifies the `Event`'s `attrName` attribute. This value may be null. `AttrChangeType` *attrChangeArg* Specifies the `Event`'s `attrChange` attribute
Returns	`void`

82

W3C NamedNodeMap interface

The `NamedNodeMap` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

`NamedNodeMap` objects in the DOM are live.

length attribute

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

`length`	
Access	read-only
Returns	`unsigned long`

getNamedItem method

Retrieves a node specified by name.

`getNamedItem`(name)	
Parameters	`String` *name* The `nodeName` of a node to retrieve.
Returns	`Node`. A `Node` (of any type) with the specified `nodeName`, or `null` if it does not identify any node in this map.

getNamedItemNS method

Retrieves a node specified by local name and namespace URI. HTML-only DOM implementations do not need to implement this method.

`getNamedItemNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the node to retrieve. `String` *localName* The local name of the node to retrieve.
Returns	`Node`. A `Node` (of any type) with the specified local name and namespace URI, or `null` if they do not identify any node in this map.

item method

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

`item`(nodeindex)	
Parameters	`unsigned long` *nodeindex* Index into this map.
Returns	`Node`. The node at the `index`th position in the map, or `null` if that is not a valid index.

removeNamedItem method

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

`removeNamedItem`(name)	
Parameters	`String` *name* The `nodeName` of the node to remove.
Returns	`Node`. The node removed from this map if a node with such a name exists.
Throws	DOMException NOT_FOUND_ERR: Raised if there is no node named `name` in this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly.

removeNamedItemNS method

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

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

`removeNamedItemNS`(namespaceURI, localName)	
Parameters	`String` *namespaceURI* The namespace URI of the node to remove. `String` *localName* The local name of the node to remove.

Returns	`Node`. The node removed from this map if a node with such a local name and namespace URI exists.
Throws	DOMException NOT_FOUND_ERR: Raised if there is no node with the specified `namespaceURI` and `localName` in this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly.

setNamedItem method

Adds a node using its `nodeName` attribute. If a node with that name is already present in this map, it is replaced by the new one.

As the `nodeName` attribute is used to derive the name which the node must be stored under, multiple nodes of certain types (those that have a "special" string value) cannot be stored as the names would clash. This is seen as preferable to allowing nodes to be aliased.

`setNamedItem`(arg)	
Parameters	`Node` *arg* A node to store in this map. The node will later be accessible using the value of its `nodeName` attribute.
Returns	`Node`. If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.
Throws	DOMException WRONG_DOCUMENT_ERR: Raised if `arg` was created from a different document than the one that created this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly. INUSE_ATTRIBUTE_ERR: Raised if `arg` is an `Attr` that is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

setNamedItemNS method

Adds a node using its `namespaceURI` and `localName`. If a node with that namespace URI and that local name is already present in this map, it is replaced by the new one.

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

`setNamedItemNS`(arg)	
Parameters	`Node` *arg* A node to store in this map. The node will later be accessible using the value of its `namespaceURI` and `localName` attributes.
Returns	`Node`. If the new `Node` replaces an existing node the replaced `Node` is returned, otherwise `null` is returned.
Throws	DOMException WRONG_DOCUMENT_ERR: Raised if `arg` was created from a different document than the one that created this map. NO_MODIFICATION_ALLOWED_ERR: Raised if this map is readonly. INUSE_ATTRIBUTE_ERR: Raised if `arg` is an `Attr` that is already an attribute of another `Element` object. The DOM user must explicitly clone `Attr` nodes to re-use them in other elements.

83

W3C NameList interface

The `NameList` interface is defined in the W3C Document Object Model (DOM) Level 3 Core Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Core.)

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

length attribute

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

`length`	
Access	read-only
Returns	`unsigned long`

contains method

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

`contains`(str)	
Parameters	`String` *str* The name to look for.
Returns	`booleantrue` if the name has been found, `false` otherwise.

containsNS method

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

`containsNS`(namespaceURI, name)	
Parameters	`String` *namespaceURI* The namespace URI to look for. `String` *name* The name to look for.
Returns	`booleantrue` if the pair namespaceURI/name has been found, `false` otherwise.

getName method

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

`getName`(index)	
Parameters	`unsigned long` *index* Index into the collection.
Returns	`String`. The name at the `index`th position in the `NameList`, or `null` if there is no name for the specified index or if the index is out of range.

getNamespaceURI method

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

`getNamespaceURI`(index)	
Parameters	`unsigned long` *index* Index into the collection.
Returns	`String`. The namespace URI at the `index`th position in the `NameList`, or `null` if there is no name for the specified index or if the index is out of range.

84

W3C Node interface

The `Node` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

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

Interface	**nodeName**	**nodeValue**	**attributes**
Attr	name of attribute	value of attribute	null
CDATASection	#cdata-section	content of the CDATA Section	null
Comment	#comment	content of the comment	null
Document	#document	null	null
DocumentFrag-ment	#document-fragment	null	null
DocumentType	document type name	null	null
Element	tag name	null	NamedNodeMap
Entity	entity name	null	null
EntityReference	name of entity referenced	null	null
Notation	notation name	null	null

Interface	nodeName	nodeValue	attributes
ProcessingInstruction	target	entire content excluding the target	null
Text	#text	content of the text node	null

NodeType enumeration

An integer indicating which type of node this is.

Note

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

The `NodeType` enumeration has the following constants of type `unsigned short`.

ELEMENT_NODE = 1
The node is an `Element`.

ATTRIBUTE_NODE = 2
The node is an `Attr`.

TEXT_NODE = 3
The node is a `Text` node.

CDATA_SECTION_NODE = 4
The node is a `CDATASection`.

ENTITY_REFERENCE_NODE = 5
The node is an `EntityReference`.

ENTITY_NODE = 6
The node is an `Entity`.

PROCESSING_INSTRUCTION_NODE = 7
The node is a `ProcessingInstruction`.

COMMENT_NODE = 8
The node is a `Comment`.

DOCUMENT_NODE = 9
The node is a `Document`.

DOCUMENT_TYPE_NODE = 10
The node is a `DocumentType`.

DOCUMENT_FRAGMENT_NODE = 11
The node is a `DocumentFragment`.

NOTATION_NODE = 12
The node is a `Notation`.

DocumentPosition enumeration

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

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

Otherwise, the order of two nodes is determined by looking for common containers – containers which contain both. A node directly contains any child nodes. A node also directly contains any other nodes attached to it such as attributes contained in an element or entities and notations contained in a document type. Nodes contained in contained nodes are also contained, but less-directly as the number of intervening containers increases.

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

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

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

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

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

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

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

The `DocumentPosition` enumeration has the following constants of type `unsigned short`.

DOCUMENT_POSITION_DISCONNECTED = 0x01
The two nodes are disconnected. Order between disconnected nodes is always implementation-specific.

DOCUMENT_POSITION_PRECEDING = 0x02
The node precedes the reference node.

DOCUMENT_POSITION_FOLLOWING = 0x04
The node follows the reference node.

DOCUMENT_POSITION_CONTAINS = 0x08
The node contains the reference node. A node which contains is always preceding, too.

DOCUMENT_POSITION_CONTAINED_BY = 0x10
The node is contained by the reference node. A node which is contained is always following, too.

DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20
The determination of preceding versus following is implementation-specific.

attributes attribute

A `NamedNodeMap` containing the attributes of this node (if it is an `Element`) or `null` otherwise.

`attributes`	
Access	read-only
Returns	`NamedNodeMap`

baseURI attribute

The absolute base URI of this node or `null` if the implementation wasn't able to obtain an absolute URI. This value is computed as described in . However, when the `Document` supports the feature "HTML" [DOM Level 2 HTML], the base

URI is computed using first the value of the href attribute of the HTML BASE element if any, and the value of the `documentURI` attribute from the `Document` interface otherwise.

`baseURI`	
Access	read-only
Returns	`String`

childNodes attribute

A `NodeList` that contains all children of this node. If there are no children, this is a `NodeList` containing no nodes.

`childNodes`	
Access	read-only
Returns	`NodeList`

firstChild attribute

The first child of this node. If there is no such node, this returns `null`.

`firstChild`	
Access	read-only
Returns	`Node`

lastChild attribute

The last child of this node. If there is no such node, this returns `null`.

`lastChild`	
Access	read-only
Returns	`Node`

localName attribute

Returns the local part of the qualified name of this node.

For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` interface, this is always `null`.

`localName`	
Access	read-only
Returns	`String`

namespaceURI attribute

The namespace URI of this node, or `null` if it is unspecified.

This is not a computed value that is the result of a namespace lookup based on an examination of the namespace declarations in scope. It is merely the namespace URI given at creation time.

For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` interface, this is always `null`.

Note

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

`namespaceURI`	
Access	read-only
Returns	`String`

nextSibling attribute

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

`nextSibling`	
Access	read-only
Returns	`Node`

nodeName attribute

The name of this node, depending on its type; see the table above.

`nodeName`	
Access	read-only
Returns	`String`

nodeType attribute

A code representing the type of the underlying object, as defined above.

nodeType	
Access	read-only
Returns	unsigned short

nodeValue attribute

The value of this node, depending on its type; see the table above. When it is defined to be null, setting it has no effect. including if the node is read-only.

nodeValue	
Access	read-write
Returns	String
Get throws	DOMException DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a DOMString variable on the implementation platform.
Set throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

ownerDocument attribute

The Document object associated with this node. This is also the Document object used to create new nodes. When this node is a Document or a DocumentType which is not used with any Document yet, this is null.

ownerDocument	
Access	read-only
Returns	Document

parentNode attribute

The parent of this node. All nodes, except Attr, Document, DocumentFragment, Entity, and Notation may have a parent. However, if a node has just been created and not yet added to the tree, or if it has been removed from the tree, this is null.

parentNode	
Access	read-only
Returns	Node

prefix attribute

The namespace prefix of this node, or `null` if it is unspecified.

Note that setting this attribute, when permitted, changes the `nodeName` attribute, which holds the qualified name, as well as the `tagName` and `name` attributes of the `Element` and `Attr` interfaces, when applicable.

Note also that changing the prefix of an attribute that is known to have a default value, does not make a new attribute with the default value and the original prefix appear, since the `namespaceURI` and `localName` do not change.

For nodes of any type other than `ELEMENT_NODE` and `ATTRIBUTE_NODE` and nodes created with a DOM Level 1 method, such as `createElement` from the `Document` interface, this is always `null`.

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

previousSibling attribute

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

`previousSibling`	
Access	read-only
Returns	`Node`

textContent attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

This attribute returns the text content of this node and its descendants. When it is defined to be `null`, setting it has no effect. On setting, any possible children this node may have are removed and, if it the new string is not empty or `null`, replaced by a single `Text` node containing the string this attribute is set to.

On getting, no serialization is performed, the returned string does not contain any markup. No whitespace normalization is performed and the returned string does not contain the white spaces in element content (see the attribute `Text.isElementContentWhitespace`). Similarly, on setting, no parsing is performed either, the input string is taken as pure textual content.

The string returned is made of the text content of this node depending on its type, as defined below:

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

`textContent`	
Access	read-write
Returns	`String`

Get throws	DOMException DOMSTRING_SIZE_ERR: Raised when it would return more characters than fit in a `DOMString` variable on the implementation platform.
Set throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

appendChild method

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

`appendChild`(newChild)	
Parameters	`Node` *newChild* The node to add. If it is a `DocumentFragment` object, the entire contents of the document fragment are moved into the child list of this node
Returns	`Node`. The node added.
Throws	DOMException HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to append is one of this node's ancestors. WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

cloneNode method

Returns a duplicate of this node, i.e., serves as a generic copy constructor for nodes. The duplicate node has no parent; (`parentNode` is `null`.).

Cloning an `Element` copies all attributes and their values, including those generated by the XML processor to represent defaulted attributes, but this method does not copy any text it contains unless it is a deep clone, since the text is contained in a child `Text` node. Cloning an `Attribute` directly, as opposed to

be cloned as part of an `Element` cloning operation, returns a specified attribute (`specified` is `true`). Cloning any other type of node simply returns a copy of this node.

Note that cloning an immutable subtree results in a mutable copy, but the children of an `EntityReference` clone are readonly. In addition, clones of unspecified `Attr` nodes are specified. And, cloning `Document`, `DocumentType`, `Entity`, and `Notation` nodes is implementation dependent.

`cloneNode`(deep)	
Parameters	`boolean` *deep* If `true`, recursively clone the subtree under the specified node; if `false`, clone only the node itself (and its attributes, if it is an `Element`).
Returns	`Node`. The duplicate node.

compareDocumentPosition method

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

`compareDocumentPosition`(other)	
Parameters	`Node` *other* The node to compare against the reference node.
Returns	`unsigned short`. Returns how the node is positioned relatively to the reference node.
Throws	DOMException NOT_SUPPORTED_ERR: when the compared nodes are from different DOM implementations that do not coordinate to return consistent implementation-specific results.

getFeature method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

`getFeature`(feature version)	
Parameters	`String` *feature* The name of the feature requested. Note that any plus sign "+" prepended to the name of the feature will be ignored since it is not significant in the context of this method. `String` *version* This is the version number of the feature to test.
Returns	`DOMObject`. Returns an object which implements the specialized APIs of the specified feature and version, if any, or `null` if there is no object which implements interfaces associated with that feature. If the `DOMObject` returned by this method implements the `Node` interface, it must delegate to the primary core `Node` and not return results inconsistent with the primary core `Node` such as attributes, childNodes, etc.

getUserData method

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

`getUserData`(key)	
Parameters	`String` *key* The key the object is associated to.
Returns	`DOMUserData`. Returns the `DOMUserData` associated to the given key on this node, or `null` if there was none.

hasAttributes method

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

`hasAttributes`( )	
Parameters	None
Returns	`trueboolean`. if this node has any attributes, `false` otherwise.

hasChildNodes method

Returns whether this node has any children.

`hasChildNodes`( )	
Parameters	None
Returns	`booleantrue` if this node has any children, `false` otherwise.

insertBefore method

Inserts the node `newChild` before the existing child node `refChild`. If `refChild` is `null`, insert `newChild` at the end of the list of children.

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

`insertBefore`(newChild [, refChild])	
Parameters	`Node` *newChild* The node to insert. `Node` *refChild* [optional] The reference node, i.e., the node before which the new node must be inserted.
Returns	`Node`. The node being inserted.
Throws	DOMException HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to insert is one of this node's ancestors. WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly or if the parent of the node being inserted is readonly. NOT_FOUND_ERR: Raised if `refChild` is not a child of this node.

isDefaultNamespace method

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

`isDefaultNamespace`(namespaceURI)	
Parameters	`String` *namespaceURI* The namespace URI to look for.
Returns	`booleantrue` if the specified `namespaceURI` is the default namespace, `false` otherwise.

isEqualNode method

Tests whether two nodes are equal.

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

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

- The two nodes are of the same type.
- The following string attributes are equal: `nodeName`, `localName`, `namespaceURI`, `prefix`, `nodeValue`, `baseURI`. This is: they are both `null`, or they have the same length and are character for character identical.
- The `attributes` `NamedNodeMaps` are equal. This is: they are both `null`, or they have the same length and for each node that exists in one map there is a node that exists in the other map and is equal, although not necessarily at the same index.
- The `childNodes` `NodeLists` are equal. This is: they are both `null`, or they have the same length and contain equal nodes at the same index. Note that normalization can affect equality; to avoid this, nodes should be normalized before being compared.

For two `DocumentType` nodes to be equal, the following conditions must also be satisfied:

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

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

`isEqualNode`(arg)	
Parameters	`Node` *arg* The node to compare equality with.
Returns	`boolean`. Returns `true` if the nodes are equal, `false` otherwise.

isSameNode method

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

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

`isSameNode`(other)	
Parameters	`Node` *other* The node to test against.
Returns	`boolean`. Returns `true` if the nodes are the same, `false` otherwise.

isSupported method

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

`isSupported`(feature, version)	
Parameters	`String` *feature* The name of the feature to test. This is the same name which can be passed to the method `hasFeature` on `DOMImplementation`. `String` *version* This is the version number of the feature to test. In Level 2, version 1, this is the string "2.0". If the version is not specified, supporting any version of the feature will cause the method to return `true`.
Returns	`boolean`. Returns `true` if the specified feature is supported on this node, `false` otherwise.

lookupNamespacePrefix method

Look up the prefix associated to the given namespace URI, starting from this node.

`lookupNamespacePrefix`(namespaceURI, useDefault)	
Parameters	`String` *namespaceURI* The namespace URI to look for. `boolean` *useDefault* Indicates if the lookup mechanism should take into account the default namespace or not.
Returns	`String`. Returns an associated namespace prefix if found, `null` if none is found and `useDefault` is false, or `null` if not found or it is the default namespace and `useDefault` is `true`. If more than one prefix are associated to the namespace prefix, the returned namespace prefix is implementation dependent.

lookupNamespaceURI method

Look up the namespace URI associated to the given prefix, starting from this node.

`lookupNamespaceURI`(prefix)	
Parameters	`String` *prefix* The prefix to look for. If this parameter is `null`, the method will return the default namespace URI if any.
Returns	`String`. Returns the associated namespace URI or `null` if none is found.

lookupPrefix method

Look up the prefix associated to the given namespace URI, starting from this node.

`lookupPrefix`(namespaceURI)	
Parameters	`String` *namespaceURI* The namespace URI to look for.
Returns	`String`. Returns an associated namespace prefix if found or `null` if none is found. If more than one prefix are associated to the namespace URI, the returned namespace prefix is implementation dependent.

normalize method

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

Note

In cases where the document contains `CDATASections`, the normalize operation alone may not be sufficient, since XPointers do not differentiate between `Text` nodes and `CDATASection` nodes.

`normalize( )`	
Parameters	None
Returns	`void`

removeChild method

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

`removeChild`(oldChild)	
Parameters	`Node` *oldChild* The node being removed.
Returns	`Node`. The node removed.
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly. NOT_FOUND_ERR: Raised if `oldChild` is not a child of this node.

replaceChild method

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

If `newChild` is a `DocumentFragment` object, `oldChild` is replaced by all of the `DocumentFragment` children, which are inserted in the same order. If the `newChild` is already in the tree, it is first removed.

`replaceChild`(newChild, oldChild)	
Parameters	`Node` *newChild* The new node to put in the child list. `Node` *oldChild* The node being replaced in the list.
Returns	`Node`. The node replaced.
Throws	DOMException HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the `newChild` node, or if the node to put in is one of this node's ancestors. WRONG_DOCUMENT_ERR: Raised if `newChild` was created from a different document than the one that created this node. NO_MODIFICATION_ALLOWED_ERR: Raised if this node or the parent of the new node is readonly. NOT_FOUND_ERR: Raised if `oldChild` is not a child of this node.

setUserData method

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

`setUserData`(key, data, handler)	
Parameters	`String` *key* The key to associate the object to. `DOMUserData` *data* The object to associate to the given key, or `null` to remove any existing association to that key. `UserDataHandler` *handler* The handler to associate to that key, or `null`.
Returns	`DOMUserData`. Returns the `DOMUserData` previously associated to the given key on this node, or `null` if there was none.

85

W3C NodeEditVAL interface

The `NodeEditVAL` interface is defined in the W3C Document Object Model (DOM) Level 3 Validation Specification. (Refer to http://www.w3.org/TR/DOM-Level-3-Val.)

This interface is similar to the [DOM Level 3 Core] `Node` interface, with methods for guided document editing.

validationState enumeration

An integer indicating the validation state, or whether the operation can or cannot be done.

The `validationState` enumeration has the following constants of type `unsigned short`.

VAL_TRUE = 5
: True if the node is valid with regards to the operation, or if the operation can be done.

VAL_FALSE = 6
: False if the node is invalid with regards to the operation, or if the operation cannot be done.

VAL_UNKNOWN = 7
: The validity of the node is unknown.

validationType enumeration

An integer indicating the validation type. Other specifications can define stricter validation types/constants by extending the `NodeEditVAL` interface.

The `validationType` enumeration has the following constants of type `unsigned short`.

VAL_WF = 1
: Check if the node is well-formed.

VAL_NS_WF = 2
: Check if the node is namespace well-formed.

VAL_INCOMPLETE = 3
: Check if the node's immediate children are those expected by the content model. This node's trailing required children could be missing. It includes `VAL_NS_WF`.

VAL_SCHEMA = 4
: Check if the node's entire subtree are those expected by the content model. It includes `VAL_NS_WF`.

defaultValue attribute

The default value specified in an attribute or an element declaration or `null` if unspecified. If the schema is a W3C XML schema, this is the canonical lexical representation of the default value.

`defaultValue`	
Access	read-only
Returns	`String`

enumeratedValues attribute

A `DOMStringList`, as described in [DOM Level 3 Core], of distinct values for an attribute or an element declaration or `null` if unspecified. If the schema is a W3C XML schema, this is a list of strings which are lexical representations corresponding to the values in the [value] property of the enumeration component for the type of the attribute or element. It is recommended that the canonical lexical representations of the values be used.

`enumeratedValues`	
Access	read-only
Returns	`DOMStringList`

canAppendChild method

Determines whether the `Node.appendChild` operation would make this document not compliant with the `VAL_INCOMPLETE` validity type.

`canAppendChild`(newChild)	
Parameters	`Node` *newChild* `Node` to be appended.
Returns	`unsigned short`. A validation state constant.

canInsertBefore method

Determines whether the `Node.insertBefore` operation would make this document not compliant with the `VAL_INCOMPLETE` validity type.

`canInsertBefore`(newChild [, refChild])	
Parameters	`Node` *newChild* `Node` to be inserted. `Node` *refChild* [optional] Reference `Node`.
Returns	`unsigned short`. A validation state constant.

canRemoveChild method

Determines whether the `Node.removeChild` operation would make this document not compliant with the `VAL_INCOMPLETE` validity type.

`canRemoveChild`(oldChild)	
Parameters	`Node` *oldChild* `Node` to be removed.
Returns	`unsigned short`. A validation state constant.

canReplaceChild method

Determines whether the `Node.replaceChild` operation would make this document not compliant with the `VAL_INCOMPLETE` validity type.

`canReplaceChild`(newChild, oldChild)	
Parameters	`Node` *newChild* New `Node`. `Node` *oldChild* `Node` to be replaced.
Returns	`unsigned short`. A validation state constant.

nodeValidity method

Determines if the node is valid relative to the validation type specified in `valType`. This operation doesn't normalize before checking if it is valid. To do so, one would need to explicitly call a normalize method. The difference between this method and the DocumentEditVAL.validateDocument method is that the latter method only checks to determine whether the entire document is valid.

`nodeValidity`(valType)	
Parameters	`unsigned short` *valType* Flag to indicate the validation type checking to be done.
Returns	`unsigned short`. A validation state constant.

86

W3C NodeList interface

The `NodeList` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

length attribute

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

`length`	
Access	read-only
Returns	`unsigned long`

item method

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

`item`(nodeindex)	
Parameters	`unsigned long` *nodeindex* Index into the collection.
Returns	`Node`. The node at the `index`th position in the `NodeList`, or `null` if that is not a valid index.

87

W3C Notation interface

The `Notation` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

The DOM Level 1 does not support editing `Notation` nodes; they are therefore readonly.

A `Notation` node does not have any parent.

publicId attribute

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

`publicId`	
Access	read-only
Returns	`String`

systemId attribute

The system identifier of this notation. If the system identifier was not specified, this is `null`.

`systemId`	
Access	read-only
Returns	`String`

88

W3C ProcessingInstruction interface

The `ProcessingInstruction` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

data attribute

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

`data`	
Access	read-write
Returns	`String`
Set throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised when the node is readonly.

target attribute

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

`target`	
Access	read-only
Returns	`String`

89

PropertyMap interface

The `PropertyMap` interface provides the abstraction of a collection of typed objects associated with string keys.

The items in the `PropertyMap` are accessible by a string key. The `keys` attribute is provided to iterate over all entries in the map.

A `PropertyMap` object can be created using the `Application.createPropertyMap` factory method. Some AOM methods return `PropertyMap` objects.

DataType enumeration

`DataType` is the return type from the `getDataType` method which identifies the type of data stored in the `PropertyMap`.

The `DataType` enumeration has the following constants of type `unsigned short`.

TYPE_UNKNOWN = 0
No data associated with the key.

TYPE_NUMBER = 1
The data value is a number.

TYPE_STRINGLIST = 2
The data value is a `StringList` object.

TYPE_STRING = 3
The data value is a `DOMString`.

keys attribute

Returns a `StringList` of all keys in the collection, which may be used to iterate over the `PropertyMap`.

`keys`	
Access	read-only
Returns	`StringList`

modified attribute

A boolean indicating whether the property map object has been modified in the current session.

`modified`	
Access	read-write
Returns	`boolean`

containsKey method

Tests whether the specified key is contained in the `PropertyMap`.

containsKey(key)	
Parameters	String *key* The key to check.
Returns	trueboolean. if the PropertyMap contains a value for key. false otherwise.

getDataType method

Returns the type of data associated with the specified key in the PropertyMap.

getDataType(key)	
Parameters	String *key* The key to examine.
Returns	unsigned short. The type associated with the key. If key is not contained in the PropertyMap, returns TYPE_UNKNOWN.

getNumber method

Returns the integer value data associated with the specified key in the PropertyMap.

getNumber(key)	
Parameters	String *key* The key to examine.
Returns	long. The numeric value associated with the key. If key is not contained in the PropertyMap or the map entry is not a number, returns -1.

getString method

Returns the string value data associated with the specified key in the PropertyMap.

getString(key)	
Parameters	String *key* The key to examine.
Returns	String. The DOMString value associated with the key. If key is not contained in the PropertyMap or the map entry is not a string, returns null.

getStringList method

Returns the `StringList` associated with the specified key in the `PropertyMap`.

`getStringList`(key)	
Parameters	`String` *key* The key to examine.
Returns	`StringList`. The `StringList` value associated with the key. If `key` is not contained in the `PropertyMap` or the map entry is not a `StringList`, returns `null`.

putNumber method

Associates a numeric value with a particular key.

`putNumber`(key, value)	
Parameters	`String` *key* Identifies the value to be replaced. `long` *value* The new value to be stored.
Returns	`void`

putString method

Associates a `DOMString` with a particular key.

`putString`(key, value)	
Parameters	`String` *key* Identifies the value to be replaced. `String` *value* The new value to be stored. If the value is `null`, the previous value, if any, is deleted from the map.
Returns	`void`

putStringList method

Associates a `StringList` with a particular key.

`putStringList`(key, value)	
Parameters	`String` *key* Identifies the value to be replaced. `StringList` *value* The new value to be stored. If the value is `null`, the previous value, if any, is deleted from the map.
Returns	`void`

remove method

Deletes an entry from the `PropertyMap`.

`remove`(key)	
Parameters	`String` *key* Identifies the item to remove. If the key is not contained in the map, does nothing.
Returns	`void`

90

W3C Range interface

The `Range` interface is defined in the W3C Document Object Model (DOM) Level 2 Traversal and Range Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113.)

CompareHow enumeration

Passed as a parameter to the `compareBoundaryPoints` method.

The `CompareHow` enumeration has the following constants of type `unsigned short`.

START_TO_START = 0
Compare start boundary-point of `sourceRange` to start boundary-point of Range on which `compareBoundaryPoints` is invoked.

START_TO_END = 1
Compare start boundary-point of `sourceRange` to end boundary-point of Range on which `compareBoundaryPoints` is invoked.

END_TO_END = 2
Compare end boundary-point of `sourceRange` to end boundary-point of Range on which `compareBoundaryPoints` is invoked.

END_TO_START = 3
Compare end boundary-point of `sourceRange` to start boundary-point of Range on which `compareBoundaryPoints` is invoked.

collapsed attribute

TRUE if the Range is collapsed

`collapsed`	
Access	read-only
Returns	`boolean`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

commonAncestorContainer attribute

The deepest common ancestor container of the Range's two boundary-points.

`commonAncestorContainer`	
Access	read-only
Returns	`Node`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

endContainer attribute

Node within which the Range ends

`endContainer`	
Access	read-only
Returns	`Node`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

endOffset attribute

Offset within the ending node of the Range.

`endOffset`	
Access	read-only
Returns	`long`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

startContainer attribute

Node within which the Range begins

`startContainer`	
Access	read-only
Returns	`Node`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

startOffset attribute

Offset within the starting node of the Range.

`startOffset`	
Access	read-only
Returns	`long`
Get throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

cloneContents method

Duplicates the contents of a Range

`cloneContents`( )	
Parameters	None
Returns	`DocumentFragment`. A DocumentFragment that contains content equivalent to this Range.
Throws	DOMException HIERARCHY_REQUEST_ERR: Raised if a DocumentType node would be extracted into the new DocumentFragment. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

cloneRange method

Produces a new Range whose boundary-points are equal to the boundary-points of the Range.

`cloneRange`( )	
Parameters	None
Returns	`Range`. The duplicated Range.
Throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

collapse method

Collapse a Range onto one of its boundary-points

`collapse`(toStart)	
Parameters	`boolean` *toStart* If TRUE, collapses the Range onto its start; if FALSE, collapses it onto its end.
Returns	`void`
Throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

compareBoundaryPoints method

Compare the boundary-points of two Ranges in a document.

`compareBoundaryPoints`(how, sourceRange)	
Parameters	`CompareHow` *how* A code representing the type of comparison, as defined above. `Range` *sourceRange* The `Range` on which this current `Range` is compared to.
Returns	`short`. -1, 0 or 1 depending on whether the corresponding boundary-point of the Range is respectively before, equal to, or after the corresponding boundary-point of `sourceRange`.
Throws	DOMException WRONG_DOCUMENT_ERR: Raised if the two Ranges are not in the same Document or DocumentFragment. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

deleteContents method

Removes the contents of a Range from the containing document or document fragment without returning a reference to the removed content.

`deleteContents`( )	
Parameters	None
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if any portion of the content of the Range is read-only or any of the nodes that contain any of the content of the Range are read-only. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

detach method

Called to indicate that the Range is no longer in use and that the implementation may relinquish any resources associated with this Range. Subsequent calls to any methods or attribute getters on this Range will result in a `DOMException` being thrown with an error code of `INVALID_STATE_ERR`.

`detach`()	
Parameters	None
Returns	`void`
Throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

extractContents method

Moves the contents of a Range from the containing document or document fragment to a new DocumentFragment.

`extractContents`( )	
Parameters	None
Returns	`DocumentFragment`. A DocumentFragment containing the extracted contents.
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if any portion of the content of the Range is read-only or any of the nodes which contain any of the content of the Range are read-only. HIERARCHY_REQUEST_ERR: Raised if a DocumentType node would be extracted into the new DocumentFragment. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

insertNode method

Inserts a node into the Document or DocumentFragment at the start of the Range. If the container is a Text node, this will be split at the start of the Range (as if the Text node's splitText method was performed at the insertion point) and the insertion will occur between the two resulting Text nodes. Adjacent Text nodes will not be automatically merged. If the node to be inserted is a DocumentFragment node, the children will be inserted rather than the DocumentFragment node itself.

`insertNode`(newNode)	
Parameters	`Node` *newNode* The node to insert at the start of the Range
Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if an ancestor container of the start of the Range is read-only. WRONG_DOCUMENT_ERR: Raised if `newNode` and the container of the start of the Range were not created from the same document. HIERARCHY_REQUEST_ERR: Raised if the container of the start of the Range is of a type that does not allow children of the type of `newNode` or if `newNode` is an ancestor of the container. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. RangeException INVALID_NODE_TYPE_ERR: Raised if `newNode` is an Attr, Entity, Notation, or Document node.

selectNode method

Select a node and its contents

`selectNode`(refNode)	
Parameters	`Node` *refNode* The node to select.
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if an ancestor of `refNode` is an Entity, Notation or DocumentType node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

selectNodeContents method

Select the contents within a node

`selectNodeContents`(refNode)	
Parameters	`Node` *refNode* Node to select from
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation or DocumentType node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setEnd method

Sets the attributes describing the end of a Range.

`setEnd`(refNode, offset)	
Parameters	`Node` *refNode* The `refNode` value. This parameter must be different from `null`. `long` *offset* The `endOffset` value.

Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation, or DocumentType node. DOMException INDEX_SIZE_ERR: Raised if `offset` is negative or greater than the number of child units in `refNode`. Child units are 16-bit units if `refNode` is a type of CharacterData node (e.g., a Text or Comment node) or a ProcessingInstruction node. Child units are Nodes in all other cases. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setEndAfter method

Sets the end of a Range to be after a node

`setEndAfter`(refNode)	
Parameters	`Node` *refNode* Range ends after `refNode`.
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode` is not an Attr, Document or DocumentFragment node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setEndBefore method

Sets the end position to be before a node.

`setEndBefore`(refNode)	
Parameters	`Node` *refNode* Range ends before `refNode`
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode` is not an Attr, Document, or DocumentFragment node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setStart method

Sets the attributes describing the start of the Range.

`setStart`(refNode, offset)	
Parameters	`Node` *refNode* The `refNode` value. This parameter must be different from `null`. `long` *offset* The `startOffset` value.

Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation, or DocumentType node. DOMException INDEX_SIZE_ERR: Raised if `offset` is negative or greater than the number of child units in `refNode`. Child units are 16-bit units if `refNode` is a type of CharacterData node (e.g., a Text or Comment node) or a ProcessingInstruction node. Child units are Nodes in all other cases. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setStartAfter method

Sets the start position to be after a node

`setStartAfter`(refNode)	
Parameters	`Node` *refNode* Range starts after `refNode`
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode` is not an Attr, Document, or DocumentFragment node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

setStartBefore method

Sets the start position to be before a node

`setStartBefore`(refNode)	
Parameters	`Node` *refNode* Range starts before `refNode`
Returns	`void`
Throws	RangeException INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode` is not an Attr, Document, or DocumentFragment node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node. DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. WRONG_DOCUMENT_ERR: Raised if `refNode` was created from a different document than the one that created this range.

surroundContents method

Reparents the contents of the Range to the given node and inserts the node at the position of the start of the Range.

`surroundContents`(newParent)	
Parameters	`Node` *newParent* The node to surround the contents with.

Returns	`void`
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if an ancestor container of either boundary-point of the Range is read-only. WRONG_DOCUMENT_ERR: Raised if `newParent` and the container of the start of the Range were not created from the same document. HIERARCHY_REQUEST_ERR: Raised if the container of the start of the Range is of a type that does not allow children of the type of `newParent` or if `newParent` is an ancestor of the container or if `node` would end up with a child node of a type not allowed by the type of `node`. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object. RangeException BAD_BOUNDARYPOINTS_ERR: Raised if the Range partially selects a non-text node. INVALID_NODE_TYPE_ERR: Raised if `node` is an Attr, Entity, DocumentType, Notation, Document, or DocumentFragment node.

toString method

Returns the contents of a Range as a string. This string contains only the data characters, not any markup.

`toString`()	
Parameters	None
Returns	`String`. The contents of the Range.
Throws	DOMException INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

91

W3C RangeException exception

The `RangeException` interface is defined in the W3C Document Object Model (DOM) Level 2 Traversal and Range Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113.)

Range operations may throw a `RangeException` as specified in their method descriptions.

Objects that implement the `RangeException` interface include the following property:

`unsigned short code`

RangeExceptionCode enumeration

An integer indicating the type of error generated.

The `RangeExceptionCode` enumeration has the following constants of type `unsigned short`.

BAD_BOUNDARYPOINTS_ERR = 1

If the boundary-points of a Range do not meet specific requirements.

INVALID_NODE_TYPE_ERR = 2

If the container of an boundary-point of a Range is being set to either a node of an invalid type or a node with an ancestor of an invalid type.

92

ScriptContext interface

The `ScriptContext` interface provides methods to load and run scripts using the Microsoft Windows Scripting engine in separate contexts.

This interface is only available in the COM binding of the AOM.

scriptType enumeration

Passed as the value of the `scriptType` parameter to `loadScriptText`.

The `scriptType` enumeration has the following constants of type `unsigned short`.

SCRIPT_GLOBAL_EXPRESSION = 0
An expression in global scope

SCRIPT_PRIVATE_EXPRESSION = 1
An expression in private scope

SCRIPT_GLOBAL_STATEMENT = 2
A statement in global scope

SCRIPT_PRIVATE_STATEMENT = 3
A statement in private scope

addTypeLibFlags enumeration

Bits defined in the `flags` parameter to `AddTypeLib`.

The `addTypeLibFlags` enumeration has the following constants of type `unsigned short`.

TYPELIB_ACTIVEX_CONTROL = 1
The type library is for an ActiveX control.

addNamedItem method

Adds a script object or COM object to the script context's variable namespace. This method makes a given script's methods available to other script instances. Such availability is important when binding events to child controls in an ActiveX component, such as to buttons residing on an HTML form launched within the Microsoft WebBrowser control. Because event binding is name-based, this method gives much greater flexibility than would normally be available from the script host.

`addNamedItem`(script, name)	
Parameters	`IDispatch` *script* The script object to define. `String` *name* The name to be defined for the object in this script context.
Returns	`void`

addTypeLib method

Adds a type library to the script context. This makes the constants defined in the library available to scripts in the context.

`addTypeLib`(progId [, version [, flags]])	
Parameters	`String` *progId* A string containing the program ID or CLSID of the object whose type library is to be added to the script context. `String` *version* [optional] The version of the type library desired. If this is not specified, the version registered for the object given in the `progId` parameter is used. If no version is registered for it, version 1.0 is assumed. The version must be a string in the form "n[.m]". `unsigned short` *flags* [optional] Flags that affect the way the type library is added to the scripting environment. Can be the sum of zero or more values from `addTypeLibFlags`.
Returns	`void`

loadScriptFile method

Loads, compiles, and runs the specified script file.

`loadScriptFile`(filename)	
Parameters	`String` *filename* The file name containing the script to load.
Returns	`IDispatch`. An object representing the compiled script. This object name can be used in the `addNamedItem` method to expose the script's methods to other script instances.

loadScriptText method

Compiles and evaluates the script expression and returns the result as a string.

`loadScriptText`(script [, scriptType])	
Parameters	`String` *script* The string containing the script. `unsigned short` *scriptType* [optional] A value from the `scriptType` definition indicating how to interpret the script text. The script can be either a statement or an expression (not all script engines make this distinction). It can also be evaluated in either private or global scope. If it is evaluated in private scope names in the script text will be discarded on return from this method. If it is evaluated in global scope top-level objects and names will persist after the call. If not provided the default is to evaluate the script as an expression in global scope.
Returns	`String`. The result value as a string if `scriptType` is an expression.

terminate method

Terminates and unloads the Microsoft Windows Script engine instance associated with this object. It gives the user the means to close a given script engine instance.

`terminate`([immediate])	
Parameters	`boolean` *immediate* [optional] If true the script context is deleted immediately, if false it is not deleted until the next message cycle, giving running scripts a chance to finish
Returns	`void`

93

StringList interface

The `StringList` interface provides the abstraction of an ordered collection of `DOMStrings`, without defining or constraining how this collection is implemented.

The items in the `StringList` are accessible by an integral index, starting from 0.

Some AOM methods return `StringList` objects. A `StringList` object can be created using the `Application.createStringList` factory method. For example,

```
var list = Application.createStringList(10);
```

creates a new `StringList` object with 10 elements, all null. The `length` attribute will return 10 in this case. To create an array where `length` returns the number of non-null entries, create the `StringList` with size 0 and add elements using the `append` method. For example,

```
var list = Application.createStringList(0);
list.append("one");
list.append("two");
list.append("three");
```

The `length` attribute would return 3 in this case.

length attribute

The current size of the list. If set to a value greater than the current size, the list is expanded with null values in the new space. If set to a smaller size, the list is truncated with the excess storage deallocated.

`length`	
Access	read-write
Returns	`unsigned long`

append method

Adds a string to the end of the current collection.

`append`(value)	
Parameters	`String` *value* New value to store into the collection, which may be `null`.
Returns	`void`

item method

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

`item`(nodeindex)	
Parameters	`unsigned long` *nodeindex* Index into the collection.
Returns	`String`. The string at the `index`th position in the `StringList`, or `null` if that is not a valid index.

setItem method

Replaces the `index`th item in the collection with a new value. If `index` is greater than or equal to the number of strings in the list, this expands the list filling with `null` values.

`setItem`(nodeindex, value)	
Parameters	`unsigned long` *nodeindex* Index into the collection. `String` *value* New value to store into the collection, which may be `null`.
Returns	`void`

94

TableCell interface

Represents a single cell in a table. May be part of a spanned `TableMulticell`, but represents a single cell in that multicell if so.

cellAbove attribute

The cell above this cell.

`cellAbove`	
Access	read-only
Returns	`TableCell`

cellBelow attribute

The cell below this cell.

`cellBelow`	
Access	read-only
Returns	`TableCell`

cellLeft attribute

The cell to the left of this cell.

`cellLeft`	
Access	read-only
Returns	`TableCell`

cellRight attribute

The cell to the right of this cell.

`cellRight`	
Access	read-only
Returns	`TableCell`

column attribute

The column that the cell is part of.

`column`	
Access	read-only
Returns	`TableColumn`

contents attribute

The contents of the cell. The `Range` returned contains the cell's contents. If the cell has no contents, a collapsed `Range` is returned. The contents of the cell can be changed by changing the contents of the `Range`.

`contents`	
Access	read-only
Returns	`Range`

multicell attribute

The `TableMulticell` that this cell is part of. Will be `null` if not part of a multicell.

`multicell`	
Access	read-only
Returns	`TableMulticell`

onBottomMulticellEdge attribute

`True` if the cell is on the bottom edge of a multicell.

`onBottomMulticellEdge`	
Access	read-only
Returns	`boolean`

onLeftMulticellEdge attribute

`True` if the cell is on the left edge of a multicell.

`onLeftMulticellEdge`	
Access	read-only
Returns	`boolean`

onRightMulticellEdge attribute

`True` if the cell is on the right edge of a multicell.

`onRightMulticellEdge`	
Access	read-only
Returns	`boolean`

onTopMulticellEdge attribute

`True` if the cell in on the top edge of a multicell.

`onTopMulticellEdge`	
Access	read-only
Returns	`boolean`

row attribute

The row that the cell is part of.

`row`	
Access	read-only
Returns	`TableRow`

ruleAbove attribute

The `TableRule` on the top edge of the cell.

`ruleAbove`	
Access	read-only
Returns	`TableRule`

ruleBelow attribute

The `TableRule` on the bottom edge of the cell.

`ruleBelow`	
Access	read-only
Returns	`TableRule`

ruleLeft attribute

The `TableRule` on the left edge of the cell.

`ruleLeft`	
Access	read-only
Returns	`TableRule`

ruleRight attribute

The `TableRule` on the right edge of the cell.

ruleRight	
Access	read-only
Returns	TableRule

spanned attribute

True if this cell is in a multicell and is not the spanning cell in the multicell

spanned	
Access	read-only
Returns	boolean

spanning attribute

True if this cell is the spanning cell in a multicell

spanning	
Access	read-only
Returns	boolean

deleteFontPI method

Deletes the font PI from the table cell if it has one. Otherwise does nothing.

deleteFontPI( )	
Parameters	None
Returns	void
Throws	TableException OPERATION_FAILED_ERR: Raised if the font PI can't be deleted.

findFontPI method

Returns an Element node for the font PI in the cell, creating it if asked to.

findFontPI(create)	
Parameters	boolean *create* If true, then the font PI is created if it doesn't exist and the table model allows the creation.

Returns	`Element`. The font PI. `Null` if none and create not specified.
Throws	TableException OPERATION_FAILED_ERR: Raised if an attempt to create font PI fails.

inSameColumn method

Returns true if this cell is in the same column in the same grid as the indicated cell.

`inSameColumn`(otherCell)	
Parameters	`TableCell` *otherCell* The other cell that may be in the same column.
Returns	`Trueboolean`. if the two cells are in the same column.

inSameRow method

Returns `true` if this cell is in the same row in the same grid as the indicated cell.

`inSameRow`(otherCell)	
Parameters	`TableCell` *otherCell* The other cell that may be in the same row.
Returns	`Trueboolean`. if the two cells are in the same row.

instantiate method

Marks the cell as being non-sparse. Some table models allow sparse markup, where some cells of a table are not explicitly described in markup. Arbortext Editor and Arbortext Editor add generated cell markup to the document when reading a sparse table so that there is markup underlying every table cell. When a document containing a table is saved, this generated markup is deleted, unless the cell has acquired content, attributes, or some other reason for existence. This function allows the user to require that the markup corresponding to a cell NOT be discarded, even if it is generated markup.

`instantiate`( )	
Parameters	None
Returns	`void`

isAdjacent method

Returns `true` if the indicated cell is a neighbor.

`isAdjacent`(otherCell)	
Parameters	`TableCell` *otherCell* The cell that might be a neighbor.
Returns	`Trueboolean`. if the other cell is a neighbor.

nextGalleyCell method

Returns the next cell in galley order, wrapping around from the last to the first if requested.

`nextGalleyCell`(wrap)	
Parameters	`boolean` *wrap* If `true`, wrap around from the last cell to the first.
Returns	`TableCell`. The next cell in galley order, if any.

previousGalleyCell method

Returns the previous cell in galley order, wrapping around from the first to the last if requested.

`previousGalleyCell`(wrap)	
Parameters	`boolean` *wrap* If `true`, wrap around from the first cell to the last.
Returns	`TableCell`. The previous cell in galley order, if any.

rectangle method

Returns a `TableRectangle` with this cell on one corner and the cell given as the parameter on the other corner. Either cell may be the upper left cell. Both must be in the same grid.

`rectangle`(otherCorner)	
Parameters	`TableCell` *otherCorner* The `TableCell` that defines the other corner of the rectangle.

Returns	`TableRectangle`. A rectangle with the specified corner cells.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the two cells are not in the same grid.

span method

Creates a span of a rectangle of cells. This cell is on one corner and the cell given as a parameter is on the other corner.

`span`(otherCorner)	
Parameters	`TableCell` *otherCorner* The cell on the other corner. Either cell may be the upper left or lower right.
Returns	`TableMulticell`. The multicell created to represent the span.
Throws	TableException INVALID_SPAN_ERR: Raised if a span can't be created from the two corner cells

unspan method

Unspans the cell, which must be in a multicell.

`unspan`()	
Parameters	None
Returns	`void`
Throws	TableException INVALID_SPAN_ERR: Raised if the cell is not in a multicell.

95

TableColumn interface

Represents either a column of cells. Every cell is part of exactly one `TableColumn`.

bottomCell attribute

The bottom cell in the column.

`bottomCell`	
Access	read-only
Returns	`TableCell`

cellCount attribute

The number of cells in the column.

`cellCount`	
Access	read-only
Returns	`unsigned long`

cells attribute

A `TableObjectStore` containing all the cells in the column.

`cells`	
Access	read-only
Returns	`TableObjectStore`

columnLeft attribute

A `TableColumn` representing the column to the left of this one. If this is the left-most column it is a null pointer.

`columnLeft`	
Access	read-only
Returns	`TableColumn`

columnRight attribute

A `TableColumn` representing the column to the right of this one. If this is the right-most column it is a null pointer.

`columnRight`	
Access	read-only
Returns	`TableColumn`

first attribute

`True` if this column is the first column in the `TableGrid`.

`first`	
Access	read-only
Returns	`boolean`

index attribute

The column number of this column in its grid. The left most column in the grid is column 1.

`index`	
Access	read-only
Returns	`unsigned long`

last attribute

`True` if this column is the last column in the `TableGrid`.

`last`	
Access	read-only
Returns	`boolean`

ruleAbove attribute

A `TableRule` for the rule at the top end of the column.

`ruleAbove`	
Access	read-only
Returns	`TableRule`

ruleBelow attribute

A `TableRule` for the rule at the bottom end of the column.

`ruleBelow`	
Access	read-only
Returns	`TableRule`

rulesLeft attribute

A `TableObjectStore` containing a `TableRule` for each rule on the left edge of this column.

`rulesLeft`	
Access	read-only
Returns	`TableObjectStore`

rulesRight attribute

A `TableObjectStore` containing a `TableRule` for each rule on the right edge of this column.

`rulesRight`	
Access	read-only
Returns	`TableObjectStore`

suppressed attribute

`True` if the entire column is suppressed because all of its cells are spanned and none of them is a spanning cell.

`suppressed`	
Access	read-only
Returns	`boolean`

topCell attribute

The top cell in the column.

`topCell`	
Access	read-only
Returns	`TableCell`

cell method

Returns a `TableCell` representing the cell at the given position in the column or a null pointer if that cell doesn't exist. The first cell is cell 1.

`cell`(cellindex)	
Parameters	`unsigned long` *cellindex* The index of the cell desired. The first cell is cell 1.

Returns	`TableCell`. The cell.
Throws	TableException INVALID_INDEX_ERR: Raised if index is less than one or greater than the number cells in the column.

96

TableException exception

Defines the exceptions used by the `Table` AOM methods.

Objects that implement the `TableException` interface include the following property:

`unsigned short code`

TableExceptionCode enumeration

An integer defining the errors generated by the `Table` AOM methods

The `TableExceptionCode` enumeration has the following constants of type `unsigned short`.

OPERATION_FAILED_ERR = 1
The operation failed because the table model did not allow it.

INVALID_INDEX_ERR = 2
An invalid row or column index, less than 1 or greater than the number of rows or columns.

INVALID_DIRECTION_ERR = 3
A direction must be 0 (right), 1 (below), 2 (left), or 3 (above).

INVALID_ORIENTATION_ERR = 4
An orientation must be 0 (vertical) or 1 (horizontal)

INVALID_SPAN_ERR = 5
Attempt to create or use an invalid cell or rule span.

INVALID_PARAMETER_ERR = 6
An invalid parameter was passed to a `table` method.

INVALID_ATTRIBUTE_ERR = 7
An invalid `table` attribute name was passed to a `table` attribute method.

97

TableGrid interface

Represents a table grid which is a rectangular array of cells. All rows and all columns are the same length.

cells attribute

A `TableObjectStore` containing all the cells in the grid. This is a static store; if cells are added to or removed from the grid (by adding or deleting rows or columns) it is not updated.

`cells`	
Access	read-only
Returns	`TableObjectStore`

columnCount attribute

The number of columns in the grid

`columnCount`	
Access	read-only
Returns	`unsigned long`

columns attribute

A `TableObjectStore` containing all the columns in the grid. This is a static store; if columns are added or removed it is not updated.

`columns`	
Access	read-only
Returns	`TableObjectStore`

firstGalleyCell attribute

The first cell in the grid in galley order.

`firstGalleyCell`	
Access	read-only
Returns	`TableCell`

gridAbove attribute

The grid above this one in the table set, if any.

`gridAbove`	
Access	read-only
Returns	`TableGrid`

gridBelow attribute

The grid below this one in the table set, if any.

`gridBelow`	
Access	read-only
Returns	`TableGrid`

index attribute

The index of this table in the `TableSet` it is part of.

`index`	
Access	read-only
Returns	`unsigned long`

lastGalleyCell attribute

The last cell in the grid in galley order.

`lastGalleyCell`	
Access	read-only
Returns	`TableCell`

rowCount attribute

The number of rows in the grid

`rowCount`	
Access	read-only
Returns	`unsigned long`

rows attribute

A `TableObjectStore` containing all the rows in the grid. This is a static store; if rows are added or removed it is not updated.

`rows`	
Access	read-only
Returns	`TableObjectStore`

rules attribute

A `TableObjectStore` containing all the rules in the grid sorted in row major order.

`rules`	
Access	read-only
Returns	`TableObjectStore`

addColumn method

Add an empty column to the grid.

`addColumn`([refColumn [, addBefore]])	
Parameters	`TableColumn` *refColumn* [optional] The column before or after which the new column is to be inserted. If `null` the new column is inserted as the last column in the grid. Some attributes of the new column are set from `refColumn`. `boolean` *addBefore* [optional] If omitted or false, the new column is added after `refColumn`, if true it is added before it.
Returns	`TableColumn`. The `TableColumn` that was inserted.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refColumn` is not in this grid or if the table model doesn't allow columns to be inserted.

addRow method

Add an empty row to the grid.

`addRow`([refRow [, addBefore]])	
Parameters	`TableRow` *refRow* [optional] The row before or after which the new row is to be inserted. If `null` the new row is inserted as the last row in the grid. Some attributes of the new row are set from `refRow` . `boolean` *addBefore* [optional] If omitted or false, the new row is added after `refRow`, if true it is added before it.

Returns	`TableRow`. The `TableRow` that was inserted.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refRow` is not in this grid or if the table model doesn't allow rows to be inserted.

cell method

Returns the cell at the specified coordinates. The upper left cell is (1,1).

`cell`(colIndex, rowIndex)	
Parameters	`unsigned long` *colIndex* The column of the cell. `unsigned long` *rowIndex* The row of the cell.
Returns	`TableCell`. The cell or `null` if no such sell exists.
Throws	TableException INVALID_INDEX_ERR: Raised if `colIndex` and `rowIndex` do not specify a cell in this grid.

column method

Returns the column given its index. The first column is column 1.

`column`(columnIndex)	
Parameters	`unsigned long` *columnIndex* The index of the column.
Returns	`TableColumn`. The indicated `TableColumn`.
Throws	TableException INVALID_INDEX_ERR: Raised if `columnIndex` does not specify a column in this grid.

deleteColumn method

Delete a column from the grid.

`deleteColumn`(column)	
Parameters	`TableColumn` *column* The column to be deleted.

Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `column` is not in this grid or if the table model does not allow it to be deleted.

deleteRow method

Delete a row from the grid.

`deleteRow`(row)	
Parameters	`TableRow` *row* The row to be deleted.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `row` is not in this grid or if the table model does not allow it to be deleted.

hlineRuleList method

Returns a table object store containing all the `TableRules` in a specified horizontal line.

`hlineRuleList`(y, startX, endX)	
Parameters	`unsigned long` *y* The vertical coordinate of the horizontal line to be returned `unsigned long` *startX* The horizontal coordinate of the left end of the rules to be returned. `unsigned long` *endX* The horizontal coordinate of the right end of the rules to be returned
Returns	`TableObjectStore`. A store containing all the rules at vertical coordinate x between `startX` and `endX`.

insertColumns method

Insert one or more columns into the grid.

`insertColumns`(contents [, refColumn])	
Parameters	`TableRectangle` *contents*

	The contents of this rectangle is copied into the new columns. This rectangle must be as high as the grid. Its width determines the number of columns inserted. `TableColumn` *refColumn* [optional] The column before which the new columns are to be inserted. If `null` the new columns are inserted as the last columns in the grid.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refColumn` is not in this grid or if the table model doesn't allow a column to be inserted.

insertRows method

Insert one or more rows into the grid.

`insertRows`(contents [, refRow])	
Parameters	`TableRectangle` *contents* The contents of this rectangle is copied into the new rows. This rectangle must be as wide as the grid. Its height determines the number of rows inserted. `TableRow` *refRow* [optional] The row before which the new rows are to be inserted. If `null` the new rows are inserted as the last rows in the grid.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refRow` is not in this grid or if the table model doesn't allow rows to be inserted.

row method

Returns a row given its index. The first row is row 1.

`row`(rowIndex)	
Parameters	`unsigned long` *rowIndex* The index of the row.

Returns	`TableRow`. The indicated `TableRow`
Throws	TableException INVALID_INDEX_ERR: Raised if `rowIndex` does not specify a row in this grid.

rule method

Returns the rule at a specified location in the grid. Rules are addressed using cell coordinates (with (1,1) being the upper left cell). For cell (m,n), (m,n) is actually the cell's upper left corner.

`rule`(startCol, startRow, endCol, endRow)	
Parameters	`unsigned long` *startCol* The starting column of the rule. `unsigned long` *startRow* The ending column of the rule. It must be `startCol` (for a vertical rule) or `startCol` + 1 (for a horizontal rule). `unsigned long` *endCol* The starting row of the rule. `unsigned long` *endRow* The ending row of the rule. It must be `startRow` (for a horizontal rule) or `starRow` + 1 (for a vertical rule).
Returns	`TableRule`. The rule at the indicated location.
Throws	TableException INVALID_INDEX_ERR: Raised if the indexes given do not specify a rule in this grid.

split method

Splits the grid at the row indicated. That row will be the top row in a new grid inserted after this one.

`split`(topRow)	
Parameters	`TableRow` *topRow* The row that should be the top row in the new grid.
Returns	`TableGrid`. The new grid that was inserted after this grid.
Throws	TableException OPERATION_FAILED_ERR: Raised if the grid can not be split because the table model does not allow multiple grids or one of the resulting grids would be invalid in some way.

vlineRuleList method

Returns a table object store containing all the `TableRules` in a specified vertical line.

`vlineRuleList`(x, startY, endY)	
Parameters	`unsigned long` *x* The horizontal coordinate of the vertical line to be returned `unsigned long` *startY* The vertical coordinate of the top end of the rules to be returned. `unsigned long` *endY* The vertical coordinate of the bottom end of the rules to be returned
Returns	`TableObjectStore`. A store containing all the rules at horizontal coordinate x between `startY` and `endY`.

98

TableMulticell interface

Represents a rectangular array of spanned cells in a table. The majority of the behavior of a `TableMulticell` is inherited from `TableRectangle`.

spanningCell attribute

The spanning cell for this multicell. This is the controlling cell for the multicell which contains all the contents of the multicell. The table model determines which cell is the spanning cell; it may be any cell in the multicell.

`spanningCell`	
Access	read-only
Returns	`TableCell`

99

TableObject interface

Base class for all `table` objects.

Type enumeration

An integer indicating which type of `table` object this is.

The `Type` enumeration has the following constants of type `short`.

INVALID_TYPE = -1
An invalid `table` object type.

TABLE_SET = 0
A table set.

TABLE_GRID = 1
A table grid.

TABLE_COLUMN = 2
A column of cells.

TABLE_ROW = 3
A row of cells.

TABLE_CELL = 4
A cell.

TABLE_RULE = 5
A rule (line) between cells.

TABLE_OBJECT_STORE = 6
A collection of `table` objects.

TABLE_TILEPLEX = 7
A table selection consisting of zero or more rectangles of cells and zero or more table rules.

Direction enumeration

A direction.

The `Direction` enumeration has the following constants of type `unsigned short`.

RIGHT = 0
To the right.

BELOW = 1
Below.

LEFT = 2
To the left.

ABOVE = 3
Above.

ExamineWhatColspec enumeration

Parameter to `renameColspec` indicating what `colspec` tags are to be examined.

The `ExamineWhatColspec` enumeration has the following constants of type `unsigned short`.

EXAMINE_ALL_COLSPECS = 0
Examine all `colspec` tags.

EXAMINE_THEAD_COLSPECS = 1
Examine `colspec` tags in a `thead`.

EXAMINE_TFOOT_COLSPECS = 2
Examine `colspec` tags in a `tfoot`.

EXAMINE_TGROUP_COLSPECS = 3
Examine `colspec` tags at the top level in a `tgroup`.

Orientation enumeration

The orientation of a rule or row/column.

The `Orientation` enumeration has the following constants of type `unsigned short`.

VERTICAL = 0
Vertical orientation. A column or vertical rule.

HORIZONTAL = 1
Horizontal orientation. A row or horizontal rule.

document attribute

The document containing this `table` object.

`document`	
Access	read-only
Returns	`Document`

element attribute

The `Element` for the markup associated with this `table` object.

`element`	
Access	read-only
Returns	`Element`

grid attribute

The `TableGrid` containing this `table` object.

`grid`	
Access	read-only
Returns	`TableGrid`

modifiable attribute

`True` if this `table` object is not read only.

`modifiable`	
Access	read-only
Returns	`boolean`

set attribute

The `TableSet` containing this `table` object.

`set`	
Access	read-only
Returns	`TableSet`

tableModel attribute

The name of the table model that manages this object. In some cases, for example if the object is a `TableObjectStore`, the table model can not be determined and `unknown` will be returned.

`tableModel`	
Access	read-only
Returns	`String`

toid attribute

The TOID of this `table` object, which is mainly useful for calling ACL routines.

`toid`	
Access	read-only
Returns	`unsigned long`

type attribute

The `TableObject.Type` (set, grid, row, column, cell, rule, and so on) of this `table` object.

`type`	
Access	read-only
Returns	`short`

clearAttributes method

Resets all of the attributes for this object to their default values and clears the corresponding table markup.

`clearAttributes`( )	
Parameters	None
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the attributes cannot be cleared.

deleteAttribute method

Delete an attribute.

`deleteAttribute`(attributeName)	
Parameters	`String` *attributeName* The `DOMString` giving the name of the attribute to be deleted.
Returns	`void`
Throws	TableException INVALID_ATTRIBUTE_ERR: Raised if the `attributeName` is not valid for this object.

deletePrivateColspecs method

All `colspec` tags within `thead` or `tfoot` tags are deleted, and every entry tag in the table is adjusted to refer to the `colspec` tags that are children of the `tgroup` element. Applied to the `TableGrid` containing the `table` object or all `TableGrid`s in the `TableSet` or `TableTilePlex` as appropriate.

`deletePrivateColspecs`( )	
Parameters	None

Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the table model doesn't support this operation or if the operation fails for some other reason.

deleteSpanspecs method

Deletes `spanspec` tags and updates entry tags to refer to `colspec` tags. If the document type does not allow `namest` or `nameend` attributes on entry elements, `deleteSpanSpecs` does nothing. This is applied to the `TableGrid` containing the `table` object or all `TableGrid`s in the `TableSet` or `TableTilePlex` as appropriate.

`deleteSpanspecs`( )	
Parameters	None
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the table model doesn't support this operation or if the operation fails for some other reason.

getAttribute method

Returns the value of an attribute given the ID of the attribute.

`getAttribute`(attributeName)	
Parameters	`String` *attributeName* The `DOMString` giving the name of the attribute.
Returns	`String`. Returns a `DOMString` representing the value for the indicated attribute.
Throws	TableException INVALID_ATTRIBUTE_ERR: Raised if the `attributeName` is not valid for this object.

minimizeAttributes method

Scans the `TableSet` containing the object and reorganizes the attributes of the various table tags to minimize the number of attributes required to describe the table.

`minimizeAttributes`( )	
Parameters	None

Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the table model doesn't support this operation or if the operation fails for some other reason.

renameColspec method

Renames a single `colspec` tag by updating the tag's `colname` attribute and adjusting all `spanspec` and `entry` tags to refer to the `colspec` tag by its the new value for the `colname` attribute. This is applied to the `TableGrid` containing the `table` object or all `TableGrid`s in the `TableSet` or `TableTilePlex` as appropriate.

`renameColspec`(colexam, oldSpecname, newSpecname)	
Parameters	`unsigned short` *colexam* An `ExamineWhatColspec` value that specifies which columns to examine while looking for this `colspec` tag. `String` *oldSpecname* A string specifying the `colspec` tag to be renamed. (That is, the `colspec` tag with `colname=oldSpecname`.) `String` *newSpecname* The new name for the `colspec` tag.
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the table model doesn't support this operation or if the operation fails for some other reason.

renameColumns method

Updates the `colname` attribute of every `colspec` tag in a table. This is applied to the `TableGrid` containing the `table` object or all `TableGrid`s in the `TableSet` or `TableTilePlex` as appropriate.

`renameColumns`(pattern, leftColspec, increment)	
Parameters	`String` *pattern* A string containing the characters `%d` specifying the root column name. (For example, "`column%d`".) `%d` is replaced with 0 for the left-most column, 1 for the next left-most, and so on. `unsigned long` *leftColspec*

	The number to be used for the left-most `colspec` tag in the table. `unsigned long` *increment* The increment to be used during the renaming process.
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the table model doesn't support this operation or if the operation fails for some other reason.

renameSpanspec method

Renames a single `spanspec` tag by adjusting the tag's `spanname` attribute and modifying every `entry` tag that refers to it. This is applied to the `TableGrid` containing the `table` object or all `TableGrid`s in the `TableSet` or `TableTilePlex` as appropriate.

`renameSpanspec`(oldSpanName, newSpanName)	
Parameters	`String` *oldSpanName* The name of the `spanspec` that is to be renamed. `String` *newSpanName* The new name for the `spanspec`.
Returns	`void`

setAttribute method

Set an attribute.

`setAttribute`(attributeName value)	
Parameters	`String` *attributeName* The `DOMString` giving the name of the attribute. `String` *value* The new value for the attribute.
Returns	`void`
Throws	TableException INVALID_ATTRIBUTE_ERR: Raised if the `attributeName` is not valid for this object or the `value` is invalid for the attribute.

100

TableObjectStore interface

A `TableObjectStore` contains a collection of `TableObjects` all from the same document. Elements can be added in any order (objects are sorted into row-major order as they are added) and retrieved through iteration.

length attribute

The number of items in the store. The valid indices are from 0 to length - 1.

`length`	
Access	read-only
Returns	`unsigned long`

addObject method

Adds a `table` object to the store.

`addObject`(item)	
Parameters	`TableObject` *item* The `table` object to be added to the store. If the object is already in the store, the request is ignored.
Returns	`void`

deleteObject method

Deletes a `table` object from the store.

`deleteObject`(item)	
Parameters	`TableObject` *item* The object to be removed from the store. The operation is ignored if the object is not in the store.
Returns	`void`

findObject method

Returns `true` if the object is in the store. `False` otherwise.

`findObject`(item)	
Parameters	`TableObject` *item* The object to look for in the store
Returns	`Trueboolean`. if the object is in the store.

item method

Returns an item from the store given its index (or `null` if the index is not valid).

`item`(itemindex)	
Parameters	`unsigned long` *itemindex* The index of the item to be returned. The first item is item 0.
Returns	`TableObject`. The requested item from the `table` object store, if any.

multicellFilter method

Create a new `table` object store from this one which contains only unspanned or spanning cells. All non-cell entries are deleted and all spanned cells are replaced by the spanning cell in the multicell (with duplicates deleted).

`multicellFilter`( )	
Parameters	None
Returns	`TableObjectStore`. A new `table` object store containing only spanning and unspanned cells

101

TableRectangle interface

Represents a rectangle of cells.

cells attribute

A `TableObjectStore` containing all the cells in the rectangle. This is static. If the `TableRectangle` changes, the contents of the `TableObjectStore` remain unchanged.

`cells`	
Access	read-only
Returns	`TableObjectStore`

cellsAbove attribute

A `table` object store containing the cells just above the rectangle.

`cellsAbove`	
Access	read-only
Returns	`TableObjectStore`

cellsBelow attribute

A `table` object store containing the cells just below the rectangle.

`cellsBelow`	
Access	read-only
Returns	`TableObjectStore`

cellsLeft attribute

A `table` object store containing the cells just to the left of the rectangle.

`cellsLeft`	
Access	read-only
Returns	`TableObjectStore`

cellsOnBottomEdge attribute

A `table` object store containing all the cells on the bottom edge of the rectangle.

`cellsOnBottomEdge`	
Access	read-only
Returns	`TableObjectStore`

cellsOnLeftEdge attribute

A `table` object store containing all the cells on the left edge of the rectangle.

`cellsOnLeftEdge`	
Access	read-only
Returns	`TableObjectStore`

cellsOnRightEdge attribute

A `table` object store containing all the cells on the right edge of the rectangle.

`cellsOnRightEdge`	
Access	read-only
Returns	`TableObjectStore`

cellsOnTopEdge attribute

A `table` object store containing all the cells on the top edge of the rectangle.

`cellsOnTopEdge`	
Access	read-only
Returns	`TableObjectStore`

cellsRight attribute

A `table` object store containing the cells just to the right of the rectangle.

`cellsRight`	
Access	read-only
Returns	`TableObjectStore`

height attribute

The height of the rectangle in rows.

`height`	
Access	read-only
Returns	`unsigned long`

lowerLeft attribute

The lower left `TableCell` in the rectangle.

`lowerLeft`	
Access	read-only
Returns	`TableCell`

lowerRight attribute

The lower right `TableCell` in the rectangle.

`lowerRight`	
Access	read-only
Returns	`TableCell`

rulesAbove attribute

A `table` object store containing the rules on the top edge of the rectangle

`rulesAbove`	
Access	read-only
Returns	`TableObjectStore`

rulesBelow attribute

A `table` object store containing the rules on the bottom edge of the rectangle

`rulesBelow`	
Access	read-only
Returns	`TableObjectStore`

rulesLeft attribute

A `table` object store containing the rules on the left edge of the rectangle

`rulesLeft`	
Access	read-only
Returns	`TableObjectStore`

rulesRight attribute

A `table` object store containing the rules on the right edge of the rectangle

`rulesRight`	
Access	read-only
Returns	`TableObjectStore`

upperLeft attribute

The upper left `TableCell` in the rectangle.

`upperLeft`	
Access	read-only
Returns	`TableCell`

upperRight attribute

The upper right `TableCell` in the rectangle.

`upperRight`	
Access	read-only
Returns	`TableCell`

valid attribute

`True` if the rectangle is valid. A rectangle may become invalid if, for example, one of its corner cells is deleted from the grid.

`valid`	
Access	read-only
Returns	`boolean`

width attribute

The width of the rectangle in columns.

`width`	
Access	read-only
Returns	`unsigned long`

copyRectangle method

Copies the contents and attributes from one rectangle to another rectangle. The two rectangles must be the same size. They do not have to be in the same document or managed by the same table model.

`copyRectangle`(sourceRectangle)	
Parameters	`TableRectangle` *sourceRectangle* The rectangle to be copied into this rectangle. It may be in a different document and it may managed by a different table model.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the source rectangle is not the same size as this rectangle. OPERATION_FAILED_ERR: Raised if the copy can not be done for a reason other than the source rectangle being a different size than this rectangle.

span method

Converts the cells in the rectangle to a spanned cell and returns the new multicell.

`span`()	
Parameters	None
Returns	`TableMulticell`. The new multicell created.
Throws	TableException INVALID_SPAN_ERR: Raised if a span can't be created from the two corner cells

102

TableRow interface

Represents a row of cells. Every cell is part of exactly one `TableRow`.

cellCount attribute

The number of cells in the row.

cellCount	
Access	read-only
Returns	unsigned long

cells attribute

A TableObjectStore containing all the cells in the row.

cells	
Access	read-only
Returns	TableObjectStore

first attribute

True if this row is the first row in the TableSet. A row that is first in the TableSet will also be first in its TableGrid. (However, a row that is first in its TableGrid will not necessarily also be first in the TableSet.)

first	
Access	read-only
Returns	boolean

index attribute

The row number of this row in its grid. The top row in the grid is row 1.

index	
Access	read-only
Returns	unsigned long

last attribute

True if this row is the last row in the TableSet. A row that is last in the TableSet will also be last in its TableGrid. (However, a row that is last in its TableGrid will not necessarily also be last in the TableSet.)

last	
Access	read-only
Returns	boolean

leftCell attribute

The left-most cell in the row.

`leftCell`	
Access	read-only
Returns	`TableCell`

rightCell attribute

The right-most cell in the row.

`rightCell`	
Access	read-only
Returns	`TableCell`

rowAbove attribute

A `TableRow` representing the row above this one. If this is the top row, it is a `null` pointer.

`rowAbove`	
Access	read-only
Returns	`TableRow`

rowBelow attribute

A `TableRow` representing the row below this one. If this is the bottom row, it is a `null` pointer.

`rowBelow`	
Access	read-only
Returns	`TableRow`

ruleLeft attribute

A `TableRule` for the rule at the left end of the row.

`ruleLeft`	
Access	read-only
Returns	`TableRule`

ruleRight attribute

A `TableRule` for the rule at the right end of the row.

`ruleRight`	
Access	read-only
Returns	`TableRule`

rulesAbove attribute

A `TableObjectStore` containing a `TableRule` for each rule on the top edge of this row.

`rulesAbove`	
Access	read-only
Returns	`TableObjectStore`

rulesBelow attribute

A `TableObjectStore` containing a `TableRule` for each rule on the bottom edge of this row.

`rulesBelow`	
Access	read-only
Returns	`TableObjectStore`

suppressed attribute

`True` if the entire row is suppressed because all of its cells are spanned and none of them is a spanning cell.

`suppressed`	
Access	read-only
Returns	`boolean`

cell method

Returns a `TableCell` representing the cell at the given position in the row or a `null` pointer if that cell does not exist. The first cell is cell 1.

`cell`(cellindex)	
Parameters	`unsigned long` *cellindex* The index of the cell desired. The first cell is cell 1.

Returns	`TableCell`. The cell.
Throws	TableException INVALID_INDEX_ERR: Raised if index is less than one or greater than the number cells in the row.

103

TableRule interface

Represents a rule. A rule is the line between two cells.

cellAbove attribute

The cell above the rule if the rule is horizontal.

`cellAbove`	
Access	read-only
Returns	`TableCell`

cellBelow attribute

The cell below the rule if the rule is horizontal

`cellBelow`	
Access	read-only
Returns	`TableCell`

cellLeft attribute

The cell to the left of the rule if the rule is vertical

`cellLeft`	
Access	read-only
Returns	`TableCell`

cellRight attribute

The cell to the right of the rule if the rule is vertical.

`cellRight`	
Access	read-only
Returns	`TableCell`

endColumnIndex attribute

The index of the ending column of the rule. This will be the number of columns plus 1 for vertical rules on the right edge of the grid.

`endColumnIndex`	
Access	read-only
Returns	`unsigned long`

endRowIndex attribute

The index of the ending row of the rule. This will be the number of rows plus 1 for horizontal rules on the bottom edge of the grid.

endRowIndex	
Access	read-only
Returns	unsigned long

orientation attribute

The orientation of the rule, that is, VERTICAL or HORIZONTAL.

orientation	
Access	read-only
Returns	Orientation

ruleAbove attribute

The rule above this rule. It will be parallel to this rule if it is a horizontal rule and join it end to end if it is a vertical rule

ruleAbove	
Access	read-only
Returns	TableRule

ruleBelow attribute

The rule below this rule. It will be parallel to this rule if it is a horizontal rule and join it end to end if it is a vertical rule

ruleBelow	
Access	read-only
Returns	TableRule

ruleLeft attribute

The rule to the left of this rule. It will be parallel to this rule if it is a vertical rule and join it end to end if it is a horizontal rule

ruleLeft	
Access	read-only
Returns	TableRule

ruleRight attribute

The rule to the right of this rule. It will be parallel to this rule if it is a vertical rule and join it end to end if it is a horizontal rule

`ruleRight`	
Access	read-only
Returns	`TableRule`

startColumnIndex attribute

The index of the starting column of the rule. This will be 1 for horizontal rules that start at the left edge of the grid or for vertical rules on the left edge of the grid.

`startColumnIndex`	
Access	read-only
Returns	`unsigned long`

startRowIndex attribute

The index of the starting row of the rule. This will be 1 for vertical rules that start at the top edge of the grid or for horizontal rules on the top edge of the grid.

`startRowIndex`	
Access	read-only
Returns	`unsigned long`

suppressed attribute

`True` if the rule is suppressed because it is inside a multicell (representing a spanned set of cells).

`suppressed`	
Access	read-only
Returns	`boolean`

104

TableSet interface

A `TableSet` is a collection of one or more `TableGrid`s, each of which is a rectangular array of `TableCell`s

gridCount attribute

The number of grids in the set.

`gridCount`	
Access	read-only
Returns	`unsigned long`

grids attribute

A list of all the `TableGrid`s in the `TableSet`.

`grids`	
Access	read-only
Returns	`TableObjectStore`

markupRange attribute

Returns a `Range` that selects all of the markup in the table.

`markupRange`	
Access	read-only
Returns	`Range`

title attribute

The table's title (or caption) for table models that define one.

`title`	
Access	read-write
Returns	`String`
Set throws	TableException OPERATION_FAILED_ERR: Raised if the insertion failed, or if the table model does not a title to be added to a set.

addGrid method

Adds an empty grid to the set.

`addGrid`(columns, rows [, refGrid [, addBefore]])	
Parameters	`unsigned long` *columns* The number of columns in the new grid.

	`unsigned long` *rows* The number of rows in the new grid. `TableGrid` *refGrid* [optional] The `TableGrid` before or after which the new one should be inserted. If this is `null`, the new grid is inserted at the end of the `TableSet`. `boolean` *addBefore* [optional] If omitted or `false`, the new grid is inserted after `refGrid`. If it is `true`, the new grid is added before `refGrid`.
Returns	`TableGrid`. The new `TableGrid`.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refGrid` is not in this set, or if the table model does not allow grids to be added to a set.

deleteGrid method

`deleteGrid`(grid)	
Parameters	`TableGrid` *grid* The `TableGrid` to be deleted from the set.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the grid is not in this set, or if the grid cannot be deleted because it is the last grid in the set, or because the table model does not allow grids to be deleted.

deleteTitle method

Delete the title for this table set.

`deleteTitle`( )	
Parameters	None
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if deletion failed, or if there is no table title to be deleted.

grid method

Returns a grid given its index. The first grid in the set is grid number 1. If no grid with this index exists, a `null` pointer is returned.

`grid`(gridindex)	
Parameters	`unsigned long` *gridindex* The index of the grid to be returned. The first grid has index 1.
Returns	`TableGrid`

insertGrid method

Inserts a new `TableGrid` into the `TableSet`.

`insertGrid`(contents [, refGrid])	
Parameters	`TableRectangle` *contents* The contents of this rectangle is copied into the new grid. `TableGrid` *refGrid* [optional] The `TableGrid` before which the new one should be inserted. If this is `null`, the new grid is inserted at the end of the `TableSet`.
Returns	`TableGrid`. The `TableGrid` added to the set.
Throws	TableException INVALID_PARAMETER_ERR: Raised if the `refGrid` is not in this set, or if the table model does not allow grids to be added to a set.

105

TableTilePlex interface

A `TableTilePlex` is used to represent a table selection. It may contain either a collection of `TableRectangle` objects or a collection of `TableRule` objects or both. All of the contents of any one tileplex must be in the same document and must be managed by the same table model.

empty attribute

`True` if the tileplex is empty.

`empty`	
Access	read-only
Returns	`boolean`

pasteRectangle attribute

If the tileplex consists of a single rectangle and no rules, this rectangle is returned. Otherwise, a `null` pointer is returned. A tileplex that contains only a single rectangle is suitable for pasting somewhere using the `TableRectangle.copyRectangle` method.

`pasteRectangle`	
Access	read-only
Returns	`TableRectangle`

valid attribute

`True` if the tileplex is valid. It is valid if all the rectangles in the tileplex are valid.

`valid`	
Access	read-only
Returns	`boolean`

addObject method

Adds a `table` object to the tileplex.

`addObject`(theObject)	
Parameters	`TableObject` *theObject* The object to be added to the tileplex. It may be a set, grid, row, column, cell, or rule. If possible, the object will be added to an existing rectangle in the tileplex. If not possible, a new rectangle will be added to the tileplex. (Unless a rule is being added.)
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the object can not be added to the tileplex because it is inconsistent with the existing contents of the tileplex.

addRectangle method

Adds a rectangle to the tileplex. It will be consolidated with an existing rectangle if possible.

`addRectangle`(theRectangle)	
Parameters	`TableRectangle` *theRectangle* The rectangle to be added to the tileplex.
Returns	`void`
Throws	TableException INVALID_PARAMETER_ERR: Raised if the rectangle can not be added to the tileplex because it is inconsistent with the existing contents of the tileplex.

clear method

Clears the tileplex by removing all rectangles and rules that the tileplex contains.

`clear`()	
Parameters	None
Returns	`void`

clonePlex method

Makes a copy of the specified tileplex. The tileplex and all of the rectangles it contains are duplicated, but the underlying `table` objects are not duplicated.

`clonePlex`( )	
Parameters	None
Returns	`TableTilePlex`. The new tileplex created by the cloning operation.

deleteFromDocument method

Deletes the contents of this tileplex from the document if possible.

`deleteFromDocument`( )	
Parameters	None
Returns	`void`
Throws	TableException OPERATION_FAILED_ERR: Raised if the contents of the tileplex can not be deleted from the document

getObjects method

Returns a `table` object store containing the contents of the tileplex interpreted according to the parameters. A given tileplex can often be interpreted in many ways. For example, as a set of grids, a set of rows, a set of columns, or a set of cells. The parameters to `getObjects` control which interpretation is desired. If it is not possible to interpret the tileplex this way, no `table` object store is returned. If several `want`xxx parameters are true, the largest possible unit (sets, grids, rows or columns, or cells) will be returned. If `wantRules` is `true`, rules will be returned if the tileplex contains any, regardless of what else is returned. If `wantRules` is false and the tileplex contains rules, nothing will be returned.

`getObjects`(wantSets, wantGrids, wantColumns, wantRows, wantCells, wantRules, contiguous, preferColumns)	
Parameters	`boolean` *wantSets* `True` if the caller will accept sets in the `table` object store returned. `boolean` *wantGrids* `True` if the caller will accept grids in the `table` object store returned. `boolean` *wantColumns* `True` if the caller will accept columns in the `table` object store returned. `boolean` *wantRows* `True` if the caller will accept rows in the `table` object store returned. `boolean` *wantCells* `True` if the caller will accept cells in the `table` object store returned. `boolean` *wantRules* `True` if the caller will accept rules in the `table` object store returned. `boolean` *contiguous* If `true`, the tileplex must cover one contiguous area in the table if anything is to be returned. `boolean` *preferColumns* When both `wantColumns` and `wantRows` are `true`, and the tileplex could be interpreted either way, return columns if `preferColumns` is `true`. Otherwise, return rows.
Returns	`TableObjectStore`. A `TableObjectStore` containing the contents of the `TableTilePlex` interpreted according to the parameters to `getObjects`.

isSelected method

Returns `true` if the tileplex selects the specified `table` object (that is, if one of the rectangles in the tileplex contains the entire rectangle defined by the `table` object).

isSelected(theObject)	
Parameters	TableObject *theObject* The set, grid, row, column, or cell that may or may not be selected by this tileplex
Returns	Trueboolean. if the TableTilePlex selects the specified table object.

pasteType method

Content that would be replaced if the tileplex were pasted to the specified location.

pasteType(targetObject)	
Parameters	TableObject *targetObject* The proposed target of the paste operation. It may be any table object so long as it is within a grid.
Returns	short. A TableObject.Type value indicating the content that would be replaced if this tileplex were pasted to the table object indicated by targetObject. If the tileplex does not contain a single rectangle, then INVALID_TYPE is returned. Otherwise TABLE_GRID, TABLE_ROW, TABLE_COLUMN, or TABLE_CELL is returned if pasting the rectangle to the indicated location would replace a grid, one or more rows, one or more columns, or a collection of cells respectively.

rectangle method

Returns the rectangle from the tileplex corresponding to the index given. The rectangles are indexed in no particular order.

rectangle(rectindex)	
Parameters	unsigned long *rectindex* The index of the rectangle to return
Returns	TableRectangle. The indicated rectangle. If no such rectangle exists, a null pointer is returned

106

W3C Text interface

The `Text` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

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

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

isElementContentWhitespace attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

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

`isElementContentWhitespace`	
Access	read-only
Returns	`boolean`

wholeText attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

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

`wholeText`	
Access	read-only
Returns	`String`

replaceWholeText method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

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

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

`replaceWholeText`(content)	
Parameters	`String` *content* The content of the replacing `Text` node.
Returns	`Text`. The `Text` node created with the specified content.
Throws	DOMException NO_MODIFICATION_ALLOWED_ERR: Raised if one of the `Text` nodes being replaced is readonly.

splitText method

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

`splitText`(offset)	
Parameters	`unsigned long` *offset* The 16-bit unit offset at which to split, starting from `0`.
Returns	`Text`. The new node, of the same type as this node.
Throws	DOMException INDEX_SIZE_ERR: Raised if the specified offset is negative or greater than the number of 16-bit units in `data`. NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.

107

ToolBarEvent interface

The `ToolBarEvent` interface provides specific contextual information associated with `ToolBar` events.

initToolBarEvent method

Initializes the value of a `ToolBarEvent` created through the `Window createEvent` method. This method should only be called before the `ToolBarEvent` has been dispatched with the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initToolBarEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

108

W3C TypeInfo interface

The `TypeInfo` interface is defined in the W3C Document Object Model (DOM) Level 2 Core Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.)

The `TypeInfo` interface represent a type referenced from `Element` or `Attr` nodes, specified in the schemas associated with the document. The type is a pair of a namespace URI and name properties, and depends on the document's schema.

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

- If this type is referenced from an `Attr` node, `typeNamespace` is `null` and `typeName` represents the [attribute type] property in the [XML Information Set]. If there is no declaration for the attribute, `typeName` is `null`.
- If this type is referenced from an `Element` node, the `typeNamespace` and `typeName` are `null`.

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

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

Note

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

- If the [validity] property exists and is "valid":

  If [member type definition] exists, then expose the {target namespace} and {name} properties of the [member type definition] property;

  If the [member type definition namespace] and the [member type definition name] exist, then expose these properties.

  If the [type definition] property exists, then expose the {target namespace} and {name} properties of the [type definition] property;

  If the [type definition namespace] and the [type definition name] exist, then expose these properties.

Note

At the time of writing, the XML Schema specification does not define how to expose anonymous types. If future specifications define how to expose anonymous types, DOM implementations can expose anonymous types via `typeName` and `typeNamespace` parameters.

Note

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

DerivationMethods enumeration

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

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

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

The `DerivationMethods` enumeration has the following constants of type `unsigned short`.

DERIVATION_EXTENSION = 1

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

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

DERIVATION_LIST = 2

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

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

DERIVATION_RESTRICTION = 3

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

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

DERIVATION_UNION = 4

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

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

typeName attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

The name of a type declared for the associated element or attribute, or `null` if unknown. Implementations may also use `null` to represent XML Schema anonymous types.

`typeName`	
Access	read-only
Returns	`String`

typeNamespace attribute

Note

This DOM Level 3 attribute is defined, but is currently unimplemented by Arbortext Editor.

The namespace of the type declared for the associated element or attribute or `null` if the element does not have declaration or if no namespace information is available. Implementations may also use `null` to represent XML Schema anonymous types.

`typeNamespace`	
Access	read-only
Returns	`String`

isDerivedFrom method

Note

This DOM Level 3 method is defined, but is currently unimplemented by Arbortext Editor.

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

`isDerivedFrom`(typeNamespaceArg, typeNameArg, derivationMethod)	
Parameters	`String` *typeNamespaceArg* Specifies the namespace of the other type definition. `String` *typeNameArg* Specifies the name of the other type definition. `unsigned long` *derivationMethod* Specifies the type of derivation and conditions applied between two types, as described in the list of constants provided in this interface.
Returns	`boolean`. If the document's schema is a DTD or no schema is associated with the document, this method will always return `false`. If the document's schema is an XML Schema, the method will return `true` if the reference type definition is derived from the other type definition according to the derivation parameter. If the value of the parameter is `0` (no bit is set to 1 for the `derivationMethod` parameter), the method will return `true` if the other type definition can be reached by recursing any combination of {base type definition}, {item type definition}, or {member type definitions} from the reference type definition.

109

W3C UIEvent interface

The `UIEvent` interface is defined in the W3C Document Object Model (DOM) Level 2 Events Specification. (Refer to http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113.)

The `UIEvent` interface provides specific contextual information associated with User Interface events.

detail attribute

Specifies some detail information about the `Event`, depending on the type of event.

`detail`	
Access	read-only
Returns	`long`

view attribute

The `view` attribute identifies the `AbstractView` from which the event was generated.

`view`	
Access	read-only
Returns	`AbstractView`

initUIEvent method

The `initUIEvent` method is used to initialize the value of a `UIEvent` created through the `DocumentEvent` interface. This method may only be called before the `UIEvent` has been dispatched via the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initUIEvent`(typeArg, canBubbleArg, cancelableArg, viewArg, detailArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented. `AbstractView` *viewArg* Specifies the `Event`'s `AbstractView`. `long` *detailArg* Specifies the `Event`'s detail.
Returns	`void`

110

View interface

The `View` interface is a subclass of `AbstractView`, representing a view of an associated `Document`. (An edit view of a document is represented as a `View` object.) An Editor frame `Window` can contain two `Views`. If a `UIEvent` is raised for a window, an event listener can use the `view` attribute of the `UIEvent` to obtain an object that implements the `View` interface (not just the `AbstractView`).

aclId attribute

An integer constant uniquely identifying the `view`. This is the value that is returned by the ACL function `current_window` if the `view` is active.

`aclId`	
Access	read-only
Returns	`long`

backgroundColor attribute

The background color of the `View`.

`backgroundColor`	
Access	read-write
Returns	`String`
Set throws	WindowException INVALID_COLOR_ERR: Raised if the `DOMString` is an unsupported color name or an invalid RGB specification.

foregroundColor attribute

The foreground color of the `View`.

`foregroundColor`	
Access	read-write
Returns	`String`
Set throws	WindowException INVALID_COLOR_ERR: Raised if the `DOMString` is an unsupported color name or an invalid RGB specification.

optionNames attribute

A `StringList` containing the names of all view-scope Arbortext `set` options.

`optionNames`	
Access	read-only
Returns	`StringList`

suspendUpdate attribute

A boolean value showing whether the view should be updated when the document is modified. Typically used when an application programmer needs to modify a large portion of the document and does not want the view to be updated until all changes have been made.

If the value is set to `true`, the view is not updated when the document is modified. If the value is set to `false`, normal updates are restored, and all changes to the document will be immediately reflected in the corresponding view. If the view is an edit view, this value only affects the modifications happened within the same script this value is set, and all edit views of the same document are affected. When the script finishes executing, the views will be updated. If the view is a dialog view, the value only affects the view it is set to, and the value affects the view until it is set to a different value .

`suspendUpdate`	
Access	read-write
Returns	`boolean`

window attribute

The `Window` in which this `view` resides.

`window`	
Access	read-only
Returns	`Window`

getOption method

This method returns the value of the Arbortext `set` option, scoped to this `view`.

`getOption`(name)	
Parameters	`String` *name* Specifies the option name, which must be a view-scope option.
Returns	`String`. The string value of the option, or null if `name` is not a valid option name. Boolean values return `on` or `off`.

setOption method

Sets the value of the Arbortext `set` option, scoped to this `view`.

`setOption`(name, value)	
Parameters	`String` *name*

	Specifies the option name, which must be a view-scope option. `String` *value* Specifies the new value of the option. Boolean values are specified using the strings `on` or `off`.
Returns	`void`
Throws	AOMException Raised if the method detects an error (for example, if `name` is not a valid view-scope option).

111

Window interface

The `Window` interface represents a top level window frame which is created by Editor.

DockEnabled enumeration

The `DockEnabled` enumeration is an integer specifying the edges of the main window this window is allowed to dock to.

The `DockEnabled` enumeration has the following constants of type `unsigned short`.

ENABLE_NONE = 0
The window is not allowed to dock.

ENABLE_TOP = 1
The window is allowed to dock at the top edge of the main window.

ENABLE_BOTTOM = 2
The window is allowed to dock at the bottom edge of the main window.

ENABLE_LEFT = 3
The window is allowed to dock at the left edge of the main window.

ENABLE_RIGHT = 4
The window is allowed to dock at the right edge of the main window.

ENABLE_TOP_BOTTOM = 5
The window is allowed to dock at the top and bottom edges of the main window.

ENABLE_TOP_LEFT = 6
The window is allowed to dock at the top and left edges of the main window.

ENABLE_TOP_RIGHT = 7
The window is allowed to dock at the top and right edges of the main window.

ENABLE_BOTTOM_LEFT = 8
The window is allowed to dock at the bottom and left edges of the main window.

ENABLE_BOTTOM_RIGHT = 9
The window is allowed to dock at the bottom and right edges of the main window.

ENABLE_LEFT_RIGHT = 10
The window is allowed to dock at the left and right edges of the main window.

ENABLE_TOP_BOTTOM_LEFT = 11
The window is allowed to dock at the top, bottom, and left edges of the main window.

ENABLE_TOP_BOTTOM_RIGHT = 12
The window is allowed to dock at the top, bottom, and right edges of the main window.

ENABLE_TOP_LEFT_RIGHT = 13
The window is allowed to dock at the top, left, and right edges of the main window.

ENABLE_BOTTOM_LEFT_RIGHT = 14
The window is allowed to dock at the bottom, left, and right edges of the main window.

ENABLE_ANY = 15
The window is allowed to dock at any edge of the main window.

DockState enumeration

The `DockState` enumeration is an integer showing the docking states of the window.

The `DockState` enumeration has the following constants of type `unsigned short`.

DOCK_NONE = 0
The window is floating.

DOCK_TOP = 1
The window is docked at the top of the main window.

DOCK_BOTTOM = 2
The window is docked at the bottom of the main window.

DOCK_LEFT = 3
The window is docked at the left of the main window.

DOCK_RIGHT = 4
The window is docked at the right of the main window.

aclId attribute

An integer constant uniquely identifying the window. This is the value that would be returned by the ACL function `current_window` if the window was active.

`aclId`	
Access	read-only
Returns	`long`

activeView attribute

A `View` object that represents the window's active view.

activeView	
Access	read-only
Returns	View

backgroundColor attribute

The background color of the window. For dialogs, you can both set and get the foreground and background colors of a window. For edit windows, you can only get the foreground and background colors of a window. You cannot set the foreground and background colors of an edit window.

backgroundColor	
Access	read-write
Returns	String
Set throws	WindowException INVALID_COLOR_ERR: Raised if the DOMString is an unsupported color name or an invalid RGB specification.

dock attribute

Indicates the docking state of the window. The value can only be changed before the window is displayed or when the window is hidden, but it can be read any time. If the value is DOCK_NONE, the window is floating.

dock	
Access	read-write
Returns	DockState
Set throws	WindowException NO_DOCKING_ALLOWED_ERR: Raised if the window is not dockable. INVALID_DOCKING_ERR: Raised if dock location is not enabled.

dockable attribute

A boolean value indicating if the window can dock to a main window.

dockable	
Access	read-only
Returns	boolean

embedded attribute

A boolean value indicating if the window frame is embedded via ActiveX into a containing parent window.

`embedded`	
Access	read-only
Returns	`boolean`

foregroundColor attribute

The foreground color of the window. For dialogs, you can both set and get the foreground and background colors of a window. For edit windows, you can only get the foreground and background colors of a window. You cannot set the foreground and background colors of an edit window.

`foregroundColor`	
Access	read-write
Returns	`String`
Set throws	WindowException INVALID_COLOR_ERR: Raised if the `DOMString` is an unsupported color name or an invalid RGB specification.

height attribute

The height of the window frame in pixels.

`height`	
Access	read-only
Returns	`int`

longNativeHandle attribute

The native window system handle associated with the window. On a Galaxy window system, this is a `vwindow` pointer.

`longNativeHandle`	
Access	read-only
Returns	`long long`

menuBar attribute

The menu bar of the window.

menuBar	
Access	read-only
Returns	MenuBar

modal attribute

A boolean value indicating if the window is modal. Modal windows grab all mouse and key events when open. The modal attribute can only be set before the window is displayed.

modal	
Access	read-write
Returns	boolean

nativeHandle attribute

The native window system handle associated with the window. On a Galaxy window system, this is a `vwindow` pointer.

This is a 32-bit value. On a 64-bit system, call getLongNativeHandle().

nativeHandle	
Access	read-only
Returns	long

optionNames attribute

A `StringList` containing the names of all window-scoped Arbortext `set` options.

optionNames	
Access	read-only
Returns	StringList

ownerNode attribute

The document `Node` that this window is associated with. This attribute will be non-null only if the window is a dialog that was created as a result of a DCF file entry that associates a dialog with a document element.

ownerNode	
Access	read-only
Returns	Node

parent attribute

The parent `Window` of this frame if it is a child window. If the `window` object is a top level window, this value is `null`. The parent attribute can only be set before the window is displayed.

`parent`	
Access	read-write
Returns	`Window`

propertyMap attribute

The `PropertyMap` associated with the window, or `null` if not set.

`propertyMap`	
Access	read-only
Returns	`PropertyMap`

screenX attribute

The X coordinate of the window frame's left edge in pixels. If the window is docked to a main window, this value is relative to the upper left corner of the dock bar.

`screenX`	
Access	read-only
Returns	`int`

screenY attribute

The Y coordinate of the window frame's top edge in pixels. If the window is docked to a main window, this value is relative to the upper left corner of the dock bar.

`screenY`	
Access	read-only
Returns	`int`

visible attribute

A boolean value indicating if the window frame is visible.

visible	
Access	read-only
Returns	boolean

width attribute

The width of the window frame in pixels.

width	
Access	read-only
Returns	int

activate method

Gives the Window focus.

activate()	
Parameters	None
Returns	void

bringToFront method

Places the Window on top of all other windows (at the top of the z-order).

bringToFront( )	
Parameters	None
Returns	void

close method

Closes this Window and releases all the native system resources it uses.

close()	
Parameters	None
Returns	void

createEvent method

Creates an event of type WindowEvent.

createEvent(eventType)	
Parameters	String *eventType*

	Specifies the type of `Event` interface to be created. The only event module supported by this method is `"WindowEvents"`. If the `Event` is to be dispatched with the `dispatchEvent` method, the appropriate event `init` method must be called after creation in order to initialize the `Event`'s values. As an example, a user wishing to synthesize a `WindowEvent` would call `createEvent` with the parameter "WindowEvents". The `initWindowEvent` method could then be called on the newly created `WindowEvent` to set the specific type of `WindowEvent` to be dispatched and to set its context information.
Returns	`Event`. The newly created `Event`.
Throws	WindowException NOT_SUPPORTED_ERR: Raised if the implementation does not support the type of `Event` interface requested.

createMenuItem method

Creates a menu item.

`createMenuItem`(label)	
Parameters	`String` *label* Specifies the label of the menu item. If this value is a dash (`-`), the method returns a menu separator (a horizontal line) that distinguishes groups of items on a submenu. Specify an access key in the label by placing an ampersand (`&`) before the character to be used as the key. For example, to specify the `F` as the access key for "File", specify the label as `&File`. The character that follows the ampersand in a label is also known as the `mnemonic` of the menu item.
Returns	`MenuItem`. The newly created `MenuItem`.

dockTo method

Docks the window to the specified location.

`dockTo`(dockState, x, y)	
Parameters	`DockState` *dockState* The manner in which the window is about to dock.

	`int` *x* The X coordinate to dock to. If the window is set to float, this value is in screen coordinates. If the window is set to dock, this value is related to the upper left corner of the dock bar to which the window docks. `int` *y* The Y coordinate to dock to. If the window is set to float, this value is in screen coordinates. If the window is set to dock, this value is related to the upper left corner of the dock bar to which the window docks.
Returns	`void`
Throws	WindowException NO_DOCKING_ALLOWED_ERR: Raised if the window is not dockable. INVALID_DOCKING_ERR: Raised if dock location is not enabled.

enableDocking method

Specifies the edges of the main window this window is allowed to dock to.

`enableDocking`(dockEnabled)	
Parameters	`DockEnabled` *dockEnabled* The edges of the main window this window is allowed to dock to.
Returns	`void`
Throws	WindowException NO_DOCKING_ALLOWED_ERR: Raised if the window is not dockable.

getOption method

Returns the value of the Arbortext `set` option, scoped to this window.

`getOption`(name)	
Parameters	`String` *name* Specifies the option name, which must be a window-scope option.
Returns	`String`. The string value of the option, or `null` if `name` is not a valid option name. Boolean values return `on` or `off`.

getScriptContext method

Returns the `ScriptContext` for the given language in this window. Returns null if there is no context for the language.

`getScriptContext`(language)	
Parameters	`String` *language* The name of the language. (For example, "VBScript" or "JScript".)
Returns	`ScriptContext`. The `ScriptContext` for the given language.

hide method

Causes the `Window` to no longer be displayed.

`hide`()	
Parameters	None
Returns	`void`

loadComponentFile method

Reads the XML file specified by `filename` and creates window components such as tool bars, menu bars, and so on according to the content of the XML File.

`loadComponentFile`(filename)	
Parameters	`String` *filename* The XML file containing the window component description. This must conform to the XML User Interface (XUI) document type.
Returns	`View`. The `View` of the new window components created by this method by using `filename`.
Throws	AOMException Raised if the method detects an error. (For example, if `filename` doesn't exist.)

moveTo method

Moves the window to the specified location.

moveTo(x, y)	
Parameters	int *x* The X coordinate to move to. A negative X value gives the X coordinate relative to the top left corner of the screen. int *y* The Y coordinate to move to. A negative Y value gives the Y coordinate relative to the top left corner of the screen.
Returns	void

sendToBack method

Places the Window behind all other windows (at the bottom of the z-order).

sendToBack( )	
Parameters	None
Returns	void

setOption method

Sets the value of the Arbortext set option, scoped to this window.

setOption(name, value)	
Parameters	String *name* Specifies the option name, which must be a window-scope option. String *value* Specifies the new value of the option. Boolean values are specified using the strings on and off.
Returns	void
Throws	AOMException Raised if the method detects an error. (For example, if name is not a valid window-scope option.)

setSize method

Changes the size of the window so it has width width and height height.

`setSize`(width, height)	
Parameters	`int` *width* The new width of the window. `int` *height* The new height of the window.
Returns	`void`

show method

Makes the `Window` visible and brings it to the front of other windows.

`show`()	
Parameters	None
Returns	`void`

112

WindowEvent interface

The `WindowEvent` interface provides specific contextual information associated with `Window` events.

initWindowEvent method

Used to initialize the value of a `WindowEvent` created through the `Window createEvent` method. This method should only be called before the `WindowEvent` has been dispatched with the `dispatchEvent` method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

`initWindowEvent`(typeArg, canBubbleArg, cancelableArg)	
Parameters	`String` *typeArg* Specifies the event type. `boolean` *canBubbleArg* Specifies whether or not the event can bubble. `boolean` *cancelableArg* Specifies whether or not the event's default action can be prevented.
Returns	`void`

113

WindowException exception

Window operations may throw a `WindowException` as specified in their method descriptions.

Objects that implement the `WindowException` interface include the following property:

`unsigned short code`

WindowExceptionCode enumeration

An integer indicating the type of error generated.

The `WindowExceptionCode` enumeration has the following constants of type `unsigned short`.

NOT_SUPPORTED_ERR = 1
The implementation does not support the requested type of object or operation.

HIERARCHY_REQUEST_ERR = 2
An attempt to insert a component in an invalid location.

WRONG_WINDOW_ERR = 3
A component is used in a window other than the one that created it (and doesn't support the component).

NOT_FOUND_ERR = 4
An attempt to reference a component or window in a context where it does not exist.

INVALID_COLOR_ERR = 5
An attempt to set color with an unsupported color name or invalid RGB specification.

INVALID_MODIFICATION_ERR = 6
An attempt to modify the type of the underlying object.

NO_MODIFICATION_ALLOWED_ERR = 7
An attempt to modify a read-only text.

NO_DOCKING_ALLOWED_ERR = 8
An attempt to dock a window which is not dockable.

INVALID_DOCKING_ERR = 9
An attempt to dock a dockable window to a main window edge which is not enabled for the dockable window.

A

AOM set Options Overview

This appendix describes the options that can be passedas the *name* parameter to the `getOption` and `setOption` methods of the followinginterfaces:

- **Application**
- **ADocument**
- **View**
- **Window**

The entire set of options that can be passed is listed in the *Arbortext Command Language Reference*.The *Arbortext Command Language Reference* is available in the Arbortext Editor Help Center. Search the HelpCenter for any option by name, or refer to the Help Center index forall options beginning with the term “ `set`”.

Options must be of the proper scope for the interface to bepassed with a method. That is, only document scope option names canbe passed with **ADocument. setOption**,only window scope option names can be passed with **Window.setOption**, and so on. The scope of each option is stated at the beginningof each option's description.

Following each option name, the allowed values are listed.

- Italics represent variable values. For example,

  `browserpath` *path*
- Curley braces represent a fixed set of possible values.For example,

  `allowinvalidmarkup` { on | off}

Option values are returned as strings by the `getOption()` methods.

Refer to the *Arbortext Command Language Reference* for a complete list of options.

Index

A

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

V

W