PTC®

PTC® Arbortext® Programmer's Reference

PTC Arbortext Editor and PTC Arbortext Publishing Engine 6.1 M040

PTC Inc., 140 Kendrick Street, Needham, MA 02494 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 PTC Arbortext Editor and the AOM to perform basic document operations and to work with events.
- *Part 4: Interfaces* — Details the W3C and PTC Arbortext interfaces (and their attributes, enumerations, and methods) supported by the AOM and the PTC Arbortext Publishing Engine.

Prerequisite Knowledge

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

Document Revision History

6.1 M040	
Event Types on page 111	Event documentation is updated with event and event type additions made over recent releases. The following event interfaces are now covered in the event documentation: • `CMSObjectEvent` • `CMSSessionConstructEvent` • `CMSSessionCreateEvent` • `CMSSessionFileEvent` • `CMSSessionBurstEvent` • `CMSAdapterConnectEvent` • `CMSAdapterDisconnectEvent`

Technical Support

Contact PTC Technical Support using the PTC website, email, phone, or fax if you encounter problems using your product or the product documentation.

Use the Contact Support links on the PTC website at:

www.ptc.com/support/

The PTC website also provides a search facility for technical documentation of particular interest. To access this search facility, use the URL above and select Search Our Knowledge.

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC Maintenance Department using the contact instructions found in your Customer Support Guide.

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 Documents

  Individual product manuals are available from the Reference Documents link of the PTC website at the following URL:

http://www.ptc.com/support/

- Help Center

  Help Centers for the most recent product releases are available from the PTC website at the URL given below. Select the Support Center for the relevant products to access the Help Centers link.

  http://www.ptc.com/support/

You must have a Service Contract Number (SCN) before you can access the Reference Documents or Help Centers links. If you do not have an SCN, contact PTC Maintenance Department 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 PTC Creo, PTC Windchill, PTC 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.

1

The *PTC Arbortext Programmer's Reference*

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**}

Where to Get More Information

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

If you're using the PTC Arbortext Publishing Engine, be sure to review *Installing PTC Arbortext Publishing Engine* and *Configuring PTC Arbortext Publishing Engine* for extensive information on PTC 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*, Second Edition, by Bruce Eckel. Published by Prentice Hall PTR. The full content of the book is available online at www.mindview.net/Books/TIJ.
- Sun has extensive Java information available at its web site java.sun.com. The tutorials are especially helpful to beginners.
- *JavaScript: The Definitive Guide*, Fourth Edition, 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.

1 Getting Started

2

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.

3

PTC Arbortext Object Model (AOM) Overview

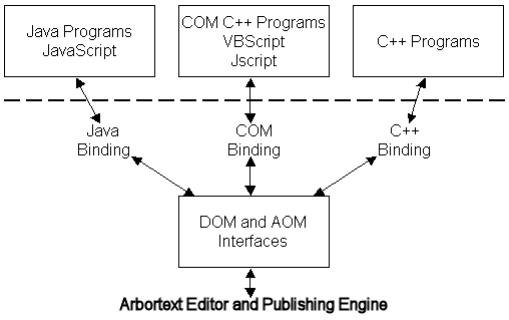
The AOM (PTC Arbortext Object Model) delivers much of ACL's functionality available to non-ACL programmers. This includes support for the W3C DOM (Document Object Model) standard. Specifically for PTC Arbortext Editor and PTC Arbortext Publishing Engine, the DOM is extended with several additional interfaces,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

Introduction to the PTC Arbortext Object Model (AOM)

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



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. PTC 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

PTC 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 PTC 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 PTC 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 PTC Arbortext Editor. Users should delete `NodeList` objects as soon after use as practical.

DOM Limitations

The PTC 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 PTC 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, PTC Arbortext

Editor inserts `xmlns` and `xmlns:prefix` XML attributes as needed to establish and maintain namespace/prefix bindings.

PTC 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.

4

Custom Applications

Overview of Custom Programs and Scripts

The PTC 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 PTC 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 PTC 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 PTC Arbortext Editor, it needs to be closed and restarted. If you're using PTC 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. PTC Arbortext Editor and PTC 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 PTC 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 *PTC Arbortext 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. PTC Arbortext Editor and PTC Arbortext Publishing Engine search each **\custom\doctypes*doctype*** directory and aggregate the list of stylesheets. For example, you can add stylesheets for the Arbortext Simplified XML DocBook Article 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, PTC 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 Arbortext Simplified XML DocBook Article **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 PTC Arbortext Editor and PTC 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 PTC Arbortext Editor and PTC 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 PTC 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 *PTC Arbortext 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 PTC 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 *PTC Arbortext 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 PTC Arbortext Styler stylesheet modules. Any modules stored in this directory are automatically available to PTC 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 **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 PTC Arbortext Editor or recorded by PTC Arbortext Publishing Engine to be sent to its HTTP client.

– **.js** (JavaScript or JScript) files

  Errors are reported to PTC Arbortext Editor or recorded by PTC 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 38 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 PTC Arbortext Editor or recorded by PTC Arbortext Publishing Engine to be sent to its HTTP client.

– **.vbs** (VBScript) files

  Errors are reported to PTC Arbortext Editor.

- **editinit** subdirectory

  Holds **.acl**, **.js**, **.class**, and **.vbs** files. Note that when you run PTC 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 PTC Arbortext Editor is not running interactively (batch mode), no errors are reported and the errors are not logged.
- PTC 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 PTC Arbortext Editor interface or held by PTC 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 *PTC Arbortext Customizer's Guide* for information on extending the PTC 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 PTC 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 PTC 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 PTC Arbortext Editor and Arbortext Publishing Engine install trees. PTC 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. PTC 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 PTC Arbortext products. At startup, PTC 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 PTC Arbortext Editor and PTC 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 for the names and descriptions of each supported subdirectory.

Note

When PTC 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 PTC 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 PTC Arbortext Editor. The other supported value is `full` meaning the application is only supported in the full installation of PTC Arbortext Editor.

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

- **Name** (required)
- **Description**
- **LicenseNumber** is only for an application distributed by PTC 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 PTC Arbortext product that supports the custom application or application. The **Product** specification helps the launching PTC 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:

  – PTC Arbortext Editor
  – Arbortext Publishing Engine
  – PTC Arbortext Architect
  – PTC Arbortext Editor with Styler

  The version must follow the convention used by PTC 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 *PTC Arbortext Customizer's Guide* for information on extending the PTC 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 PTC 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 *Configuring PTC 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.

PTC 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 PTC 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 PTC 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, PTC Arbortext Editor unzips those customizations to a directory in the PTC Arbortext Editor cache directory (**.aptcache\zc**). At start up, PTC Arbortext Editor checks to see whether the zip file has been updated. If it has, PTC Arbortext Editor downloads and uncompresses the updated customizations. If not, PTC 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 PTC 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, PTC 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 PTC 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, PTC 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

5

Using ACL with the AOM

You can access the PTC 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 PTC 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 PTC Arbortext Editor Help Center for the proper letter case for the **Acl** methods.*

6

Using Java to Access the AOM

Java Interface Overview

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor or the PTC Arbortext Publishing Engine.

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC Arbortext Publishing Engine, the JNI interacts specifically with the AOM.

PTC Arbortext Editor or an Arbortext PE sub-process creates only one instance of the embedded JVM per session and initializes it the first time a Java method is executed. The **–js** startup option may be specified when launching PTC 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 PTC 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 44.

Java Interface Platform Requirements

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

Refer to the *Installing PTC Arbortext Editor, PTC Arbortext Styler, and PTC Arbortext Architect* or *Installing PTC Arbortext Publishing Engine* for the most recent version support information. To use a different JRE, set the ACL **set** option **javavmpath** or Advanced Preference to the location of the alternate JVM.

PTC Arbortext Editor and the PTC Arbortext Publishing Engine make no attempt to use any other JVM installed on your system, even if it is already loaded for use by another program. If you want to use another 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 **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.PTC 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, PTC 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, PTC 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, PTC Arbortext Editor loads its embedded Java Virtual Machine (JVM). You can also load the embedded JVM using the **java_init** function. The embedded JVM is dedicated to running Java code started from within PTC Arbortext Editor. PTC Arbortext Editor creates only one instance of the embedded JVM per session. The JVM is unloaded when you end the current PTC Arbortext Editor session.

PTC Arbortext Editor makes no attempt to use a public JVM installed on your system, even if it is already loaded for use with another program. If you choose to load another JVM, specify it with the **set javavmpath** ACL command. To set the maximum size of the Java Virtual Machine (JVM) memory allocation pool, use the **set javavmmemory** ACL command.

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

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 PTC 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 PTC Arbortext Editor class path.

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

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

PTC Arbortext Editor and the PTC 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.

Package	Description
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 PTC 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 100 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 17 Interface Overview on page 167.

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 in the embedded JVM. Java programs running in a JVM outside of PTC 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

The simplest way to make your classes available to PTC 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 PTC 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-PTC 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 embedded Java Virtual Machine has started, changes to the **javaclasspath** option or to the directories it specifies will not take affect until you exit and start a new session of PTC Arbortext Editor or stop and restart the servlet container for the PTC Arbortext Publishing Engine.

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 PTC 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 17 Interface Overview on page 167 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

Because the PTC Arbortext embedded JVM supports Sun's Java Platform Debugger Architecture (JPDA, see http://java.sun.com/products/jpda/),any JPDA compliant Java debugger can hook to PTC Arbortext's embedded JVM.

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 Sun JDK version 1.5.0 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 PTC Arbortext Publishing Engine and PTC 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 PTC 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 PTC 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 PTC 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 PTC 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.

7

Using JavaScript to Access the AOM

JavaScript Interface Overview

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC Arbortext Publishing Engine.

PTC 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).

PTC 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 PTC 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 62 for details.

Note

The PTC Arbortext Editor JavaScript implementation supports the DOM and PTC 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. PTC 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 44 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`**. PTC 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 PTC Arbortext Editor or the Arbortext PE sub-process executes one of these ACL functions or commands or reads a **`.js`** customization file. PTC 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 PTC 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 57 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 PTC 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 **www.mozilla.org/rhino/limits.html** for additional limitations of the interpreter, and *Tutorial: Embedding Rhino* at the Mozilla web page **www.mozilla.org/rhino/tutorial.html** 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 PTC 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 PTC 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. ***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.*

Function	Description
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 173. If ***filename*** is not found relative to the current directory and is not an absolute path, the list of directories specified in the PTC Arbortext Editor (or the PTC Arbortext Publishing Engine) *loadpath* parameter is searched to locate the JavaScript source file.
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 PTC Arbortext Editor and the PTC 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 PTC 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 PTC 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 (PTC Arbortext Command Language).

Object	Description
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.
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 PTC Arbortext Editor provides a mechanism called LiveConnect that lets you use Java classes and methods from JavaScript. The PTC 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 59.

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 PTC 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 173) 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 PTC 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 PTC Arbortext distribution. Refer to Sample JavaScript Code on page 65 for details.

JavaScript Interface Error Handling

Errors When Executing JavaScript

When executing JavaScript programs, PTC 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, PTC 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 51 and defined in 17 Interface Overview on page 167. 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 61. 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

PTC 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.

8

Using COM to access the AOM

COM Interface Overview

PTC 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 PTC Arbortext Editor.

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

When acting as a COM server, PTC 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 PTC Arbortext Editor COM interface to the DOM portion of AOM uses the COM binding defined by Microsoft with changes for DOM Level 2 and PTC Arbortext extensions. However, Microsoft has made several significant extensions to the DOM that are not supported by PTC Arbortext. The definition of the COM classes and methods that PTC 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 PTC Arbortext Editor or PTC 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.

PTC Arbortext Editor or an Arbortext PE sub-process does not need to be running for it to be available to COM. If PTC Arbortext Editor or an Arbortext PE sub-process is not running when a call is made to the PTC 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 PTC 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 PTC Arbortext Editor as a COM Server

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

Starting with release 5.4, PTC Arbortext Editor also automatically checks at startup to see whether the application is registered as a COM server. If PTC Arbortext Editor finds that it is not registered as a COM server, it performs a COM registration for PTC 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), PTC Arbortext Editor still opens but displays an error message first saying that this version is no longer configured correctly. In this case, some PTC Arbortext Editor components might not be available. You can keep PTC Arbortext Editor from opening in this case with the **APTFAILIFNOCOM** environment variable.

If you run a version of PTC 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 PTC 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 PTC 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\x64\editor.exe -RegServer
Arbortext-path\bin\x86\editor.exe -RegServer
```

Registers a specific PTC Arbortext Editor installation as a COM server.

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

Unregisters a specific PTC 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 PTC Arbortext Editor installation you are running matches the PTC Arbortext Editor installation listed as the current `editor.exe` COM server.

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

Unregisters any version of PTC 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 PTC 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 PTC 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 PTC 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 PTC 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 PTC 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 69).

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*.

PTC 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.

PTC Arbortext Editor and PTC 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 PTC 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 17 Interface Overview on page 167 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

Error Code	Value
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_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

Error Code	Value
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 79 and VBScript Error Handling on page 84 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.

9

Using JScript to Access the AOM

JScript Interface Overview

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC Arbortext Publishing Engine.

PTC 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). PTC 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* *on page* .

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`**. PTC 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 PTC Arbortext Editor or the Arbortext PE sub-process executes one of these ACL functions or commands or reads a **`.js`** customization file. PTC 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 PTC 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 77 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 PTC 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 on page and on page for more information.

JScript Global Objects

The PTC 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 PTC 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 71 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

PTC 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.

10

Using VBScript to Access the AOM

VBScript Interface Overview

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC Arbortext Publishing Engine.

PTC 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). PTC 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 PTC 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 on page .*

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 PTC 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 on page and on page for more information.

VBScript Global Objects

The PTC 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 PTC 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 71 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

11

Overview of Programming and Scripting Techniques

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

- Overview on page 90 — Contains a series of examples demonstrating basic techniques for manipulating documents and content using the DOM and AOM.
- Overview on page 100 — Summarizes the DOM Event Model interfaces and the AOM extended event interfaces supported by PTC Arbortext Editor and the PTC Arbortext Publishing Engine.
- Working with Tables Overview on page 138 — The AOM contains interfaces that provide access to more than 100 PTC Arbortext Editor table functions. This chapter provides several examples that illustrate the basics of inserting and manipulating tables using the interfaces.
- Overview on page 146 — XSL composition refers to PTC 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 152 — 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 PTC Arbortext distributed document type, and detailed instructions for building your own.

12

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 PTC Arbortext XML Docbook sample opened with PTC Arbortext Editor. (Choose **File ▸New**, check **Sample**, select **PTC Arbortext XML Docbook V4.0**, and click **OK**.) Example code that calls **openDocument** requires access to one or two saved copies of the PTC 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 on page 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 "`Gem&uuml;tlichkeit`". 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");
```

13

Events

Overview

PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC 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 associated with **Activex** events.

Interface	Description
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.
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. PTC Arbortext Editor and the PTC Arbortext Publishing Engine support all of the DOM Level 2 event modules except **HTMLEvents**. In addition, PTC Arbortext Editor and the PTC 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 PTC 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 PTC 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 PTC Arbortext Dynamic Link Manager. It is a Java-only implementation that is documented in the Javadoc available in the PTC 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 PTC Arbortext Editor and the PTC 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.

Event	Occurrence
	• 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.
`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 111 for a description of each event type.

Notes and Limitations

The following notes and limitations apply to the PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor or the PTC 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 PTC Arbortext Editor and the PTC 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 PTC Arbortext Editor and the PTC 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 PTC 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 PTC 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 PTC 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 PTC 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 PTC 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 PTC 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`

DLMEvent Module

The **DLMEvent** module has the following event types. Refer also to the PTC Arbortext Dynamic Link Manager Javadoc available in the PTC Arbortext Editor Help Center.

DLMAfterDocValidation
: This event is fired immediately after the links in a document or region are validated. It can be used to perform any cleanup necessary. The target of the event is the **DLMDocument** whose links are being validated.

- Related Object: Null
- Related Node: The DOM **Document**.
- Bubbles: Yes
- Cancelable: No

DLMBeforeDocValidation
: This event is fired immediately before the links in a document or region are validated. It can be used to perform any overall set up, such as cache initialization, before validation begins. The target of the event is the **DLMDocument** whose links are being validated.

- Related Object: Null
- Related Node: The DOM **Document**.
- Bubbles: Yes
- Cancelable: No

DLMIdAssignment
: This event is fired immediately before PTC Arbortext Dynamic Link Manager assigns an XML ID to a target node in a document. The target of this event is the **DLMDocument** object containing the element being assigned an XML ID. Note that canceling this event during registration effectively cancels the registration of the target. This event bubbles from the **DLMDocument** to the **DLMClient**.

- Related Object: None
- Related Node: The element being assigned an ID value.
- Bubbles: Yes

- Cancelable: Yes

DLMBeforeTargetRegistered

This event is fired before a target from a PTC Arbortext Dynamic Link Manager document is added to a registration pass. Note that this is also called for **atidlm:link** nodes, since the source resource of all of a link's resource pairs is the link markup itself. Canceling the event for **atidlm:link** elements means that the link represented by the markup will not be considered for registration. The target for this event is the **DLMTarget** being registered.

The **DLMTarget** instance attached to the event will be discarded after registration is completed, so attaching event listeners to this object should be avoided.

- Related Object: The **DLMTarget**.
- Related Node: The target element.
- Bubbles: Yes
- Cancelable: Yes

DLMBeforeLinkRegistered

This event is fired before a link in a PTC Arbortext Dynamic Link Manager document is added to a registration pass. The target for this event is the **DLMLink** being registered.

The **DLMLink** instance attached to the event will be discarded after registration is completed, so attaching event listeners to this object should be avoided.

- Related Object: The **DLMLink**.
- Related Node: The **atidlm:link** element.
- Bubbles: Yes
- Cancelable: Yes

DLMLinkCreated

This event is fired when a **DLMDocument** creates a new **DLMLink** object. The target for this event is the **DMLink** object. Because the link is not yet present in the document, there is no node associated with this event.

- Related Object: The **DLMLink**.
- Related Node: None
- Bubbles: Yes
- Cancelable: No

DLMTargetCreated

This event is fired when a **DLMSession** creates a new **DLMTarget** object. The target for this event is the **DLMTarget** object.

- Related Object: The **DLMTarget**.
- Related Node: None
- Bubbles: Yes

- Cancelable: No

DLMLinkCommittingToDocument

This event is fired when a **DLMLink** is updated within a document's markup. Note that if the link is being inserted into the document for the first time, as opposed to being updated, the related node will be null.

- Related Object: The **DLMLink**.
- Related Node: The link node if present. Otherwise, null.
- Bubbles: Yes
- Cancelable: No

DLMTargetCommitted

This event is fired when a **DLMTarget** has its changes committed to the repository (most often from DLM Explorer). The target of this event is the target being committed. Note that this event is not fired during document registration.

- Related Object: The **DLMTarget**.
- Related Node: None
- Bubbles: Yes
- Cancelable: Yes

DLMLogin

This event is fired immediately after a session is established with PTC Arbortext Dynamic Link Manager. The target of this event is the new **DLMSession**.

- Related Object: None
- Related Node: None
- Bubbles: Yes
- Cancelable: No

DLMLogout

This event is fired immediately before a session is closed with PTC Arbortext Dynamic Link Manager. The target of this event is the **DLMSession** being closed.

- Related Object: None
- Related Node: None
- Bubbles: Yes
- Cancelable: No

DLMWorkOffline

This event is fired when a user enters offline mode with PTC Arbortext Dynamic Link Manager. The target of this event is the **DLMClient** object.

- Related Object: None
- Related Node: None
- Bubbles: Yes

- Cancelable: No

14

Working with Tables

Working with Tables Overview

The AOM contains interfaces that provide access to more than 100 PTC Arbortext Editor table functions. With these interfaces, you can programmatically create and modify tables in any PTC Arbortext Editor document using Java, JavaScript, VB, or VBScript. The entire PTC 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 PTC 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 PTC 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 PTC Arbortext Editor, open a PTC Arbortext XML Docbook template, and enter the following commands at the PTC 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 PTC Arbortext Editor, open a PTC Arbortext XML Docbook template, insert a 5x5 table, and enter the following command at the PTC Arbortext Editor command line:

   ```
   source insertcol.js
   ```

3. Select a portion of the table.
4. Enter the following command at the PTC 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 PTC 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 PTC Arbortext Editor, open a PTC Arbortext XML Docbook or an XHTML v1.0 template, and enter the following commands at the PTC 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
```

15

Working with XSL Composition

Overview

XSL composition refers to PTC 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 PTC Arbortext Architect (**Edit ▸CCF**). Several **`.ccf`** files are distributed with PTC 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();
  }
  }
    }

}
```

16

Line Numbering in PTC Arbortext Editor and the PTC Arbortext Publishing Engine

Line Numbering Overview

PTC Arbortext Editor and the PTC 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

PTC Arbortext Editor and PTC 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 PTC 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 PTC 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 PTC 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 PTC Arbortext Editor or PTC 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.
- Documents with line numbering applied cannot be checked into the Documentum XML repository if a validating application is being used on the Documentum XML side. The Documentum XML parser does not recognize namespaces.

The following limitations apply to the sample application, but are not necessarily limitations of the PTC 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 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 160.

The `atipl` markup is added to your document.

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.

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 PTC 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 160. 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 PTC 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 PTC 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 PTC 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

17

Interface Overview

The AOM supports most of the DOM interfaces developed by the W3C, several PTC Arbortext extensions to the DOM interfaces, and many additional PTC Arbortext interfaces for features that are not part of the DOM. Refer to Introduction to the Document Object Model (DOM) on page 18 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 PTC Arbortext extension to the W3C DOM **Document** interface.
ADocumentType	PTC Arbortext extensions to the W3C DOM **DocumentType** interface
AEditEvent	Provides specific contextual information associated with the **EditEvent** extension.
AElement	The PTC Arbortext extension to the W3C DOM **Element** interface.
AEvent	The PTC Arbortext extension to the W3C DOM **Event** interface.
ANode	The PTC Arbortext extension to the W3C DOM **Node** interface.
Application	Provides access to PTC Arbortext Editor and PTC 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 PTC 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.
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).

Interface	Description
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.
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.

Interface	Description
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.
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.

Interface	Description
TableTilePlex	Used to represent a table selection.
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 PTC 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 PTC Arbortext Publishing Engine server.
CCCompositionParameter	Describes a single parameter to a PTC Arbortext Content Pipeline composer (`.ccf` file).
CCDoctype	Describes a single document type installed on a PTC Arbortext Publishing Engine server.
CCDocumentComposer	Describes a composer associated with a document type installed on a PTC Arbortext Publishing Engine server.
CCFrameset	Describes a frameset that is installed on a PTC Arbortext Publishing Engine server.
CCPathEntry	Describes a single directory on a server path list.
CCStylesheet	Describes a stylesheet installed on the PTC Arbortext Publishing Engine server.
CompositionConfiguration	Provides information about a PTC Arbortext Publishing Engine server's composition capabilities.
E3Application	Creates an object that runs in each Arbortext PE sub-process and is called by the PTC Arbortext Publishing Engine to process HTTP requests.
E3ApplicationRequest	Provides request information for a PTC Arbortext Publishing Engine Application.

Interface	Description
E3ApplicationResponse	Provides an object to assist a PTC 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 PTC Arbortext Publishing Engine Composition Client.
E3Config	Passes information to a PTC Arbortext Publishing Engine Application during initialization.
E3ServerComposer	Describes an object that handles composition operations on a PTC Arbortext Publishing Engine server. “Composition" includes transforming an input JAR file into an output JAR file.
E3ServerCompositionExtension	Extends the PTC 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 PTC Arbortext Publishing Engine server composition application.
E3ServerCompositionResult	Describes the result of a composition operation under the PTC Arbortext Publishing Engine server composition application.
E3Tracer	Creates entries in the PTC Arbortext Publishing Engine Server Composition trace files.

AOM set Options A

AOM set Options Overview

This appendix describes the options that can be passed as the *name* parameter to the **getOption** and **setOption** methods of the following interfaces:

- **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 PTC Arbortext Editor Help Center in PDF and HTML forms. Search the Help Center for any option by name, or refer to the Help Center index for all options beginning with the term "`set`".

Options must be of the proper scope for the interface to be passed with a method. That is, only document scope option names can be 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 beginning of 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