Powerful VCL/CLX toolkit transforms and manipulates XML data like magic!

XMLPartner™ *Professional*

The all-in-one toolkit for reading, writing, manipulating, transforming, and presenting XML

- *Transform data into an endless variety of formats using new XSL Processor with XPath support.*
- *Create HTML, RTF, and printed documents with new rendering filters. No need to write translation code from scratch.*
- *Create, edit, and test XML and XSL fast with EXMLPro utility*

TURBOPOWER®
Software Company

XMLPartner Professional™

TurboPower Software Company
Colorado Springs, CO

www.turbopower.com

© 2001 TurboPower Software Company. All rights reserved.

License Agreement

This software and its documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

© 2001 by TurboPower Software Company. All rights reserved.

TurboPower Software Company authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without run-time fees or further licenses, your own compiled programs based on any of the source code of XMLPartner and XML Partner Professional. You may not distribute any of the XMLPartner or XMLPartner Professional source code, compiled units, or compiled example programs without written permission from TurboPower Software Company. You may not use XMLPartner or XMLPartner Professional to create components or controls to be used by other developers without written approval from TurboPower Software Company.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend upon XMLPartner or XMLPartner Professional. However, others who receive your source code or units need to purchase their own copies of XMLPartner or XMLPartner Professional in order to compile the source code or to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. Group programming projects making use of this software must purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software Company for volume discounts and site licensing agreements.

This software and accompanying documentation is deemed to be “commercial software” and “commercial computer software documentation,” respectively, pursuant to DFAR Section 227.7202 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display or disclosure of the Software by the US Government or any of its agencies shall be governed solely by the terms of this agreement and shall be prohibited except to the extent expressly permitted by the terms of this agreement. TurboPower Software Company, PO Box 49009, Colorado Springs, CO 80949-9009.

With respect to the physical media and documentation provided with XMLPartner and XMLPartner Professional, TurboPower Software Company warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software Company will replace the defective media or documentation at no cost to you.

TurboPower Software Company warrants that the software will function as described in this documentation for a period of 60 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software Company will update the defective software at no cost to you.

TurboPower Software Company further warrants that the purchaser will remain fully satisfied with the product for a period of 60 days from receipt. If you are dissatisfied for any reason, and TurboPower Software Company cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software Company, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software Company honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase a product directly from us.

TURBOPOWER SOFTWARE COMPANY DOES NOT ASSUME ANY LIABILITY FOR THE USE OF XMLPartner AND XMLPartner Professional BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE COMPANY BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section and to any additional licensing terms contained in the DEPLOY.HLP file. If you do not agree, you should immediately return the entire XMLPartner or XMLPartner Professional package for a refund.

All TurboPower product names are trademarks or registered trademarks of TurboPower Software Company. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Table of Contents i

Chapter 1: Introduction 1
Before You Begin Using XMLPartner 3
What's New in This Release? 4
Conformance 7
System Requirements 8
Files Supplied 9
The XMLPartner Class Hierarchy 13
Installation 15
Organization of this Manual 17
Technical Support 20
Suggested Reading 21

Chapter 2: An Overview of XML 23
XML Syntax 25
Using XML 38
The Advantages of XMLPartner 41

Chapter 3: Event-Driven Parsing 43
Character Sets 44
Using This Component 45
Handling Errors 50
Memory Usage 51
TXpParser Component 52

Chapter 4: Document Object Model Core Interfaces 75
TXpObjModel Component 85
TXpDOMException Class 101
TXpDOMImplementation Class 104
TXpDocumentFragment Class 107
TXpNode Class 108
TXpDocument Class 127
TXpNodeList Class 139
TXpNamedNodeMap Class 145
TXpCharacterData Class 150
TXpAttribute Class 154
TXpElement Class 156
TXpText Class 175

TXpComment Class 178

Chapter 5: Document Object Model Extended Interfaces 179
TXpCDATASection Class 182
TXpDocumentType Class 183
TXpDTDElement Class 187
TXpDTDElementGroup Class 189
TXpDTDElementContent Class 192
TXpDTDAttList Class 196
TXpDTDAttDefinition Class 200
TXpDTDEntity Class 204
TXpDTDNotation Class 208
TXpDTDConditional Class 210
TXpEntityReference Class 212
TXpProcessingInstruction Class 213

Chapter 6: An Overview of XSL 215
Creating a Stylesheet 217
Programming a Stylesheet 230

Chapter 7: XSL Processor & Filter Reference 239
The Result Tree 240
Using the XSL Components 241
TXpXSLProcessor Component 244
Filters 262
TXpFilterBase Class 263
TXpFilterHTML Component 272
TXpFilterPrint Component 276
TXpFilterRTF Component 280
TXpFilterXML Component 282

Chapter 8: XSLT Reference 285
XPath Expressions 286
Patterns 293
Attribute Value Templates 294
XSLT Element Reference 295

Chapter 9: XSL-Formatting Objects and Properties 355
XSL Area Model 357
Page Sequences and Layout 358
Formatting Content 362

Chapter 10: Appendix 461

DOM Validation Error Codes 462
Exception Error Codes 464
XPath Functions 466
Predefined colors for formatting properties 468
XSLT and XSL-FO tag codes 474
XSL-FO Property Tag Codes 477
Glossary 481

Identifier Index i

Subject Index ix

Chapter 1: Introduction

XMLPartner helps add the power of XML and XSL to Delphi, C++ Builder, and Kylix products. XMLPartner comes in two versions. The first, XMLPartner, gives you the choice of processing documents with an event-driven model or Document Object Model (DOM). XML documents loaded into the DOM may be queried using XPath 1.0 or validated against the document's Document Type Definition (DTD). In addition, XMLPartner provides full support for namespaces.

The second version, XMLPartner Professional, includes all the features provided by XMLPartner as well as support for Extensible Stylesheet Language (XSL) and Extensible Stylesheet Language Transformations (XSLT). XMLPartner Professional provides an XSL processor to support transformation of your XML documents into forms required by other applications. The processor also supports reformatting and output of the transformed documents to HTML, Print Preview, Rich Text Format (RTF), ASCII, and XML.

The XMLPartner product line provides native VCL and CLX components that support the creation, modification, parsing, and transformation of XML documents. XMLPartner compiles into your executables, making it easier to deploy, reducing the size of your installations, and helping you to avoid DLL conflicts.

Written with the world in mind, the XMLPartner product line processes documents encoded in the ISO-8859-1, UTF8, UTF16, UTF16LE, and UTF16BE character sets. This means your applications can parse and create XML documents written in European and Asian languages.

XMLPartner's parser uses a configurable, buffered stream to process XML documents instead of reading the entire document into memory at once. This means that your products will be able to handle very large documents with a minimum of overhead. Documents and external entities may be loaded and parsed whether they reside in memory, on a local or network hard drive, an HTTP URL, or FTP site. Newly created or modified documents may be saved to memory, a local or network hard drive, an HTTP URL, or FTP site.

XMLPartner's DOM implementation is one of the most complete implementations available. While the XMLPartner parser is not a validating parser, the XMLPartner DOM allows validation of a document against its Document Type Definition (DTD). If you need to search an XML document for information, the DOM provides you with the full power of the XML Path Language (XPath) 1.0. XMLPartner also conforms to the W3C Namespace recommendation.

XMLPartner Professional includes an XSL processor and several filters that aid in the transformation and formatting of your XML documents. This allows your programs to turn an XML document into a completely different XML document or into some other medium

suitable for display or import into another program. For example, a document created by a company's documentation team could be transformed into an HTML page for display on the Web or to RTF for output as a hard copy manual.

The XMLPartner product line also includes a sample XML Editor. The XML Editor shows how to display the full hierarchy of any XML document using the XMLPartner DOM. In addition to viewing documents, the XML Editor can also be used to create XML documents from scratch or make changes to existing documents. The XML Editor provided with XMLPartner Professional also allows you to preview an XML documented after it has been processed with an XSL stylesheet.

Before You Begin Using XMLPartner

If you have purchased XMLPartner, please be aware that Chapters 6 through 9 of this manual are specific to XMLPartner Professional. Chapters 2 through 5 apply to both XMLPartner and XMLPartner Professional.

In order to get the most out of XMLPartner, it is best that you have some familiarity with XML. Chapter 2 of this manual provides a brief overview of XML, event-driven parsing, the Document Object Model (DOM), and ideas for using XMLPartner. If this is your first experience with XML, we recommend that you read Chapter 2 and then take advantage of the large amount of information available on the Internet and in technical books. The "Suggested Reading" section on page 21 lists a number of valuable XML resources.

To learn how to use the event-driven parser, read the first few pages of "Chapter 3: Event-Driven Parsing" on page 43. For an introduction to the DOM component, read the first few pages of "Chapter 4: Document Object Model Core Interfaces" on page 75.

Once you are familiarized with XML and wish to take advantage of the features provided in XMLPartner Professional, read Chapter 6 in order to gain a basic understanding of stylesheets and XSLT. Read Chapter 7 to learn how to use the XSL processor and filters. The subsequent chapters serve as a reference for the XSLT elements and formatting objects supported by XMLPartner Professional.

What's New in This Release?

The following features have been added since the previous release of XMLPartner:

Support for large XML documents

In the previous version, XMLPartner would load an entire XML document into memory including any external entities referenced by that document. Depending upon the workstation's memory and swap space, attempting to process a very large document could result in a program crash.

XMLPartner now reads all XML documents, whether stored on file or in a string, using a buffered stream. Only a portion of each document resides in memory at any given point in time meaning that your products can handle several large documents at once.

Recognition of UTF8 and UTF16 character sets

XMLPartner now recognizes the UTF8, UTF16, UTF16BE, and UTF16LE character sets. The parser automatically detects the character set when the document is first opened. Unrecognized character sets are automatically treated as though they were UTF8. XMLPartner handles the situation where a document references one or more external entities that are coded in differing character sets.

Implementation of XPath 1.0

The DOM implementation now supports expressions following the XML Path Language (XPath) 1.0 recommendation. This means that you can search for data within an XML document using characteristics such as position within the document, element name, and attribute values. XMLPartner also supports the standard XPath functions and allows you to support and define your own functions.

Support for DOM Level 2 Core

XMLPartner now supports most of the DOM Core Level 2 specification. At the time of this writing, the following list identifies the Level 2 types, methods, and attributes that are not supported (review the README.HLP file for an updated list).

- DOMTimeStamp
- Document.GetElementByID
- NamedNodeMap.SetNamedItemNS
- DocumentType.InternalSubset
- Element.HasAttributeNS

Representation of Document Type Definition (DTD)

The XMLPartner DOM implementation now represents an internal or external DTD in the document tree. XMLPartner declares specific classes for element, attribute, notation, and conditional declarations. These classes are specific to XMLPartner and are not part of the DOM specification.

You can use the DTD classes to evaluate the DTD of an existing document, modify an existing DTD, or build a DTD from scratch. When the document is written to a file, the DTD is written in a well-formed manner.

Support for namespaces

The XMLPartner DOM implementation conforms to the W3C Namespace recommendation. XML namespaces provide you with a way to qualify element and attribute names within an XML document, avoiding collisions between like-named elements and attributes that have totally different meanings.

Validation of documents

The XMLPartner parser is a non-validating parser. However, if you load an XML document using the XMLPartner DOM implementation then you can validate the contents of the document against its Document Type Definition (DTD).

Support for XSLT 1.0 and XSL 1.0

The XSL processor supplied with XMLPartner Professional supports the XSLT 1.0 specification and a subset of the XSL 1.0 specification. XSLT is a powerful, programming-like, scripting language for making full use of XML documents. Transformed XML documents may be output to ASCII, HTML, Printer, RTF, and XML.

Kylix support

XMLPartner may be used with Borland Kylix for Linux development. For specific information regarding Kylix support, please see the README file included with the installation.

Conformance

XMLPartner supports the DOM Level 1 Core as well as a majority of the DOM Level 2 Core. XMLPartner's event-driven parser does not support the SAX model. XMLPartner supports XML Namespaces 1.0 and XPath 1.0. However, XMLPartner's XPath implementation does not handle infinity, positive zero, negative zero, or NaN (Not a Number).

XMLPartner Professional supports the same specifications as XMLPartner. In addition, it supports all of the elements defined in XSLT 1.0 and a subset of the formatting objects and properties defined in XSL 1.0. Deviations from the XSLT specification are documented in Chapter 8 on an element-by-element basis.

System Requirements

The following requirements are in effect for the Windows version of XMLPartner:

1. A computer capable of running Windows 9*x*, NT, or 2000. A minimum of 32MB of RAM is required.
2. Delphi version 3.0 or later, or C++Builder version 3.0 or later.
3. A hard disk with at least 20 MB of free space. To install all XMLPartner files and compile the example programs requires at least 10 MB.
4. To retrieve XML documents via HTTP or FTP, or to save XML documents to an FTP site, WININET.DLL must be installed on the workstation. WININET.DLL is supplied with Microsoft Internet Explorer.

Files Supplied

XMLPartner includes units and programs in source, precompiled, and executable form. It also includes example programs and a help system. Some files may not be installed: Setup installs files based upon your choices during installation. If there is a path prefixed to the file name, it is found from the XMLPartner installation directory (\TurboPower\XPPRO by default for XMLPartner Professional or \TurboPower\XPSTD by default for XMLPartner Standard).

Help files

README.HLP

A standard help file that describes the changes to the documentation and new features added since this manual was printed. Please read this file before using XMLPartner.

HELP*.HLP

The XMLPartner Delphi and C++Builder help files.

Source files

XpAbout.pas, XpAboutw.pas

Contains the TurboPower About box displayed for the Version property of each component and within the example XML editor. Unit XpAbout is the Kylix version of the About box and XpAboutw is the Windows version of the About box.

XpBase.pas

Declares base types and constants. Also declares utility routines for character conversion.

XpChrFlt.pas

Defines the classes for filtering XML documents.

XpDOM.pas

Defines the classes comprising the XMLPartner Document Object Model (DOM) implementation.

XpExcept.inc

Defines the strings associated with exceptions raised by XMLPartner. This file contains resource strings for a number of languages including English, German, Spanish, French, Norwegian, Swedish, and Danish.

XpExcept.pas

Defines the XMLPartner exception classes.

XpFltBas.pas

This unit is provided with XMLPartner Professional only. It defines an abstract filter class that serves as the foundation for custom filters as well as the filters delivered with XMLPartner Professional.

XpFltHTM.pas

This unit is available only in XMLPartner Professional. It defines the filter responsible for HTML output.

XpFltPrt.pas

This unit is available only in XMLPartner Professional. It defines the filter responsible for Printer (i.e., Print Preview) output.

XpFltRTF.pas

This unit is available only in XMLPartner Professional. It defines the filter responsible for Rich Text Format (RTF) output.

XpFltXML.pas

This unit is available only in XMLPartner Professional. It defines the filter responsible for XML output.

XpInet.pas

Contains routines supporting retrieval of documents via HTTP and FTP, as well as saving of documents via FTP.

XpParser.pas

Defines the event-driven parsing classes.

XpReg.pas

Contains code that registers the XMLPartner and XMLPartner Professional components with the C++Builder and Delphi Integrated Development Environments (IDEs).

XpXSLPrc.pas

This unit is available only in XMLPartner Professional. It defines the XSL processor that supports transformation and formatting of XML documents.

Packages

To avoid version conflicts with applications using different versions of the XMLPartner packages, each new version of XMLPartner installs packages using slightly different names. TurboPower's current package naming convention is as follows:

- The first letter represents one of the TurboPower products (X for XMLPartner).
- The next three digits are the product version number (250 for 2.50).
- Two product-specific digits (PR for XMLPartner Professional run-time, PD for XMLPartner Professional design or SR for XMLPartner Standard run-time, SD for XMLPartner Standard design).
- The last two digits represent the VCL version (30, 35, 40, 41, 50, 51).

The XMLPartner Professional run-time packages and design packages for Delphi 3/4/5 and C++Builder 3/4/5 are:

```
X250PR30.DPL, X250PD30.DPL          Delphi 3
X250PR40.BPL, X250PD40.BPL          Delphi 4
X250PR50.BPL, X250PD50.BPL          Delphi 5
X250PR35.BPL, X250PD35.BPL          C++Builder 3
X250PR41.BPK, X250PD41.BPL          C++Builder 4
X250PR51.BPL, X250PD51.BPL          C++Builder 5
```

The XMLPartner run-time packages and design packages for Delphi 3/4/5 and C++Builder 3/4/5 are:

```
X250SR30.DPL, X250SD30.DPL          Delphi 3
X250SR40.BPL, X250SD40.BPL          Delphi 4
X250SR50.BPL, X250SD50.BPL          Delphi 5
X250SR35.BPL, X250SD35.BPL          C++Builder 3
X250SR41.BPK, X250SD41.BPL          C++Builder 4
X250SR51.BPL, X250SD51.BPL          C++Builder 5
```

Example folders

The example programs are organized into subfolders, according to the IDE for which the example is targeted.

EXAMPLES\CBUILDRX*.*

The XMLPartner examples for C++Builder where X is the compiler version.

EXAMPLES\DELPHI*.*

The XMLPartner Standard example files for the supported versions of Delphi.

EXAMPLES\XMLDOCS*.*

Sample XML documents for use with the examples.

Examples supplied

The following examples are supplied with XMLPartner and XMLPartner Professional.

TextView

Uses XML to display formatted text in a TRichEdit component. This example is not installed with the Kylix version of XMLPartner because the TRichEdit component is unavailable in Borland Kylix for Linux.

XMLEdit

Displays an XML document in a tree view, allowing you to browse, modify, and save the document. You may also use this example to create XML documents from scratch.

XMLTest

Uses the XML Parser to verify an XML document is well-formed. It also shows you how to perform interactive queries against an XML document loaded into the DOM.

The XMLPartner Class Hierarchy

Exception (VCL/CLX)
- TXpDOMException (XPDOM)

TComponent (VCL/CLX)
- TXpParser (XPParse)
- TXpObjModel (XPDOM)

TCustomControl (VCL/CLX)
- TXpFilterBase (XPFLTBAS - Professional only)
  - TXpFilterHTML (XPFLTHTM - Professional only)
  - TXpFilterPrint (XPFLTPRT - Professional only)
  - TXpFilterRTF (XPFLTRTF - Professional only)
  - TXpFilterXML (XPFLTXML - Professional only)
- TXpXSLProcessor (XPXSLPRC - Professional only)

TObject (VCL/CLX)
- TXpDOMImplementation (XPDOM)
- TXpNode (XPDOM)
  - TXpAttribute (XPDOM)
  - TXpCharacterData (XPDOM)
    - TXpComment (XPDOM)
    - TXpText (XPDOM)
      - TXpCDATASection (XPDOM)
  - TXpDocument (XPDOM)
  - TXpDocumentFragment (XPDOM)
  - TXpDocumentType (XPDOM)
    - TXpDTDConditional (XPDOM)
  - TXpDTDAttDefinition (XPDOM)
  - TXpDTDAttList (XPDOM)

- TXpDTDElement (XPDOM)
- TXpDTDElementContent (XPDOM)
- TXpDTDElementGroup (XPDOM)
- TXpDTDEntity (XPDOM)
- TXpDTDNotation (XPDOM)
- TXpElement (XPDOM)
- TXpEntityReference (XPDOM)
- TXpProcessingInstruction (XPDOM)

TXpNodeList (XPDOM)

TXpNamedNodeMap (XPDOM)

Installation

Install XMLPartner or XMLPartner Professional directly from the TurboPower Product Suite CD. Simply insert the CD into your drive, select the appropriate product from the list of products, click “Install”, and follow the instructions. If the TurboPower introductory splash screen does not appear automatically upon insertion of the CD, run *X*:\CDROM.EXE where *X* is the letter of your CD drive.

IDE integration for XMLPartner Professional

The installation program should integrate XMLPartner properly within each of the IDEs you selected during installation. If it cannot find one of the IDEs, you will have to install XMLPartner manually. The following steps assume XMLPartner is installed on your disk.

1. Ensure XMLPartner packages are in the directory you selected during installation, \TurboPower\XPRO by default. The files you need depend on you which IDE you’re using. Use the following chart to see which files you need for XMLPartner Professional.

IDE	Packages Required
Delphi 3	X250PR30.BPL, X250PD30.BPL
Delphi 4	X250PR40.BPL, X250PD40.BPL
Delphi 5	X250PR50.BPL, X250PD50.BPL
C++ Builder 3	X250PR35.BPL, X250PD35.BPL
C++ Builder 4	X250PR41.BPL, X250PD41.BPL
C++ Builder 5	X250PR51.BPL, X250PD51.BPL

2. Start your IDE.
3. Add the XMLPartner Professional design-time package to the component package list by going to Component | Install | Packages on the menu. When it asks you to select a package, select the filename containing PD and your version of XMLPartner Professional based on the previous chart.

IDE Integration fo XMLPartner Standard

The installation program should integrate XMLPartner properly within each of the IDEs you selected during installation. If it cannot find one of the IDEs, you will have to install XMLPartner manually. The following steps assume XMLPartner is installed on your disk.

1. Ensure XMLPartner packages are in the XMLPartner directory you selected during installation, \TurboPower\XPSTD by default. The files you need depend on you which IDE you're using. Use the following table to see which files you need.

Table 1.1: *The packages required for IDE integration*

IDE	Packages required
Delphi 3	X250SR30.BPL, X250SD30.BPL
Delphi 4	X250SR40.BPL, X250SD40.BPL
Delphi 5	X250SR50.BPL, X250SD50.BPL
C++ Builder 3	X250SR35.BPL, X250SD35.BPL
C++ Builder 4	X250SR41.BPL, X250SD41.BPL
C++ Builder 5	X250SR51.BPL, X250SD51.BPL

2. Start your IDE.
3. Add the XMLPartner design-time package to the component package list by going to Component | Install | Packages on the menu. When prompted to select a package, select the filename containing SD and your version of XMLPartner based on Table 1.1.

Organization of this Manual

This manual is organized as follows:

- Chapter 1 introduces XMLPartner Professional and discusses its installation.
- Chapter 2 provides a basic overview of XML, event-driven parsing, and the Document Object Model (DOM).
- Chapter 3 describes the XMLPartner event-driven parser.
- Chapter 4 describes the XMLPartner Document Object Model core interfaces.
- Chapter 5 discusses the XMLPartner Document Object Model extended interfaces.
- Chapter 6 provides a basic overview of XSL, showing you how to create and program stylesheets. This chapter applies to XMLPartner Professional only.
- Chapter 7 describes the XSL processor and filters. This chapter applies to XMLPartner Professional only.
- Chapter 8 describes the XSLT elements supported by XMLPartner Professional.
- Chapter 9 describes the XSL formatting objects and properties supported by XMLPartner Professional.
- Chapter 10 is an Appendix that lists the error messages raised by the parser and DOM, lists the various constants defined by XMLPartner, lists the XPath functions supported by XMLPartner, and contains a glossary of common XML terms.

An identifier index and a conventional subject index are provided.

Class reference chapters start with an overview of the classes and components discussed in that chapter. The overview also includes a hierarchy for those classes and components. Each class and component is then documented individually, in the following format:

Overview

Describes the class or unit.

Hierarchy

Shows the ancestors of the class being described, generally stopping at a VCL/CLX class. The hierarchy also lists the unit in which each class is declared and the number of the first page of the documentation of each ancestor. Some classes in the hierarchy are identified with a

number in a bullet: ❶. This indicates that some of the properties, methods, or events listed for the class being described are inherited from this ancestor and documented in the ancestor class.

Properties

Lists all the properties in the class. Some properties may be identified with a number in a bullet: ❶. These properties are documented in the ancestor class from which they are inherited.

Methods

Lists all the methods in the class. Some methods may be identified with a number in a bullet: ❶. These methods are documented in the ancestor class from which they are inherited.

Events

Lists all the events in the unit. Some events may be identified with a number in a bullet: ❶. These events are documented in the ancestor class from which they are inherited.

Reference Section

Details the properties, methods, and events. These descriptions are in alphabetical order. They have the following format:

- Declaration of the property, method, or event.
- Default value for properties, if appropriate.
- A short, one-sentence purpose. A ↳symbol is used to mark the purpose to make it easy to skim through these descriptions.
- Description of the property, method, or event. Parameters are also described here.
- Examples are provided in many cases.
- The “See also” section lists other properties, methods, or events that are pertinent to this item.

Throughout the manual, the 💣 symbol is used to mark a warning or caution. Please pay special attention to these items.

Naming conventions

To avoid class name conflicts with components and classes included with Delphi, C++Builder, Kylix, or from other third party suppliers, all XMLPartner class names begin with “TXp”. The “Xp” stands for XMLPartner.

On-line help

Although this manual provides a complete discussion of each component, keep in mind there is an alternative source of information available. Once properly installed, help is available from within IDEs. Pressing <F1> with the caret on an XMLPartner Standard property or component or from within Object Inspector displays help for the item.

Technical Support

The best way to get an answer to your technical support question is to post it in the XMLPartner newsgroup on our news server (news.turbopower.com). Many of our customers find the newsgroups a valuable resource where they can learn from others' experiences and share ideas in addition to getting quick answers to questions.

To get the most from the newsgroups, it is recommended that you use dedicated newsreader software.

Newsgroups are public, so please do not post your product serial number, product unlocking code, or any other private numbers (such as credit card numbers) in your messages.

TurboPower's KnowledgeBase is another excellent support option. It has hundreds of articles about TurboPower products accessible through an easy-to-use search engine (www.turbopower.com/search). The KnowledgeBase is open 24 hours a day, 7 days a week so you'll have another way to find answers to your questions even when we're not available.

Other support options are described in the support brochure included with XMLPartner Standard. You can also read about support options at www.turbopower.com/support.

Suggested Reading

For the official definition of XML syntax, we suggest you read the *Extensible Markup Language (XML) 1.0 Recommendation*, available at http://www.w3c.org.

For help interpreting the XML specification, we recommend *XML: The Annotated Specification* by Bob DuCharme (Prentice Hall PTR).

For an introduction to XML and DOM, a useful book to read is *Beginning XML* by David Hunter (Wrox Press Ltd.).

For the official DOM interface definitions, read the *Document Object Model (DOM) Level 2 Specification*, available at http://www.w3c.org.

The official XPath specification, *XML Path Language (XPath) Version 1.0*, is also located at http://www.w3c.org.

To better understand namespaces, review the *Namespaces in XML* recommendation, available at http://www.w3c.org.

The official XSL specification, *Extensible Stylesheet Language (XSL) Version 1.0*, is available at http://www.w3c.org/TR/xsl.

The official XSLT specification, *XSL Transformations (XSLT) Version 1.0*, is available at http://www.w3c.org.

The W3Schools.com Web site, http://www.w3schools.com, provides helpful tutorials on XML, DOM, and XSL.

For an exhaustive set of information regarding XSLT and XPath functions, see *XSLT Programmer's Reference* by Michael Kay (Wrox Press Ltd.).

Chapter 2: An Overview of XML

The acronym XML stands for EXtensible Markup Language. *Markup* means the adding of codes to information in order to define the structure of the information and the manner in which the information is to appear. Before electronic publishing, copy editors would write markup (i.e., instructions) on manuscripts so that typesetters would know how to format the text.

The introduction of computers led to people using word processing software for the creation and editing of documents. Some of the earlier word processing applications required the person to enter markup codes (e.g., bold this text, indent this paragraph) directly into the text. Subsequent word processing applications began hiding the markup behind the scenes. WordPerfect and Microsoft Word are good examples. When viewing a document, you see text as it would be printed on paper. However, the word processor maintains a data structure containing the text and markup codes influencing the display and printing of the text.

The problem with this situation is that information produced by most word processors is not directly interchangeable. The desire for a universally interchangeable document format led to the creation of the Standard Generalized Markup Language (SGML). SGML is a means of formally describing a markup language as opposed to being a markup language. It allows the creation of documents that can be transported from one hardware or software platform to another without loss of information. One of the most pervasive and widely known uses of SGML is HyperText Markup Language (HTML).

What does all of that have to do with XML? XML was born out of the history of markup, SGML, and HTML. Simply put, XML was created because it solves a problem that HTML and SGML couldn't solve.

The problem was that people want to share information electronically across the Web. HTML, as seen by its widespread use, is a very good medium for displaying information. However, the HTML markup doesn't tell a software application what that information is. When viewing a Web page, you might see a company name at the top of the page, but there is nothing in the Web page markup stating, "This is the name of a company." You discern the meaning by what you see and by the context of the information on the Web page. A software application doesn't have your brain. Another issue with HTML is that its markup has a fixed vocabulary. You can't make a Web page tell a software application the meaning of information contained within the page.

Something that can define the meaning of information was needed. The markup used had to be extendible so that any kind of information could be shared. SGML seemed to fit the bill. However, some of its features made it more complicated than necessary for this task and would be difficult or costly to implement in a software application. A group of World Wide

Web Consortium (W3C) representatives formed to investigate the problem concluded that SGML would not work for the Web. They eventually developed XML, which is a subset of the SGML language.

As stated in the Extensible Markup Language (XML) 1.0 recommendation, the goal of XML "is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML."

XML is a language for defining and transporting information. While HTML's focus is displaying information, XML's focus is describing information. Like SGML, XML is a standard for defining the meaning of markup within a document. When you write an XML document, you must supply *tags* (i.e., markup) that have meaning. Tags are not predefined like they are in HTML. XML is not a replacement for HTML. Instead, the two technologies are complimentary. HTML can be used to display XML content. As laid out in the W3C recommendation, XML handles European and Asian character sets. So it is truly a universal format for exchanging information.

XML doesn't necessarily make life easy. It's a tool that requires some intelligence. It does not remove the pain of having to describe your data. It requires as much forethought as when you first design a database schema. Just as a poorly designed set of database tables leads to headaches and unforeseen changes, a poorly designed set of XML information leads to the same thing.

XML Syntax

An XML document is text-based and human-readable. You can use a text editor and some Web browsers to view an XML document. As explained in the previous section, XML is a language for defining the meaning of information. As such, it provides you with rules for defining and organizing information within a document, but puts the burden of defining markup on your shoulders. The goal of this section is to provide you with a brief overview of XML rules and syntax so that you can create simple XML documents and understand common facets of XML.

XML syntax is not that complicated. An XML document begins with a prolog. However, the prolog is not required. Following is the prolog for a sample document that is constructed in this chapter:

```
<?xml version="1.0"?>
```

If a document contains a prolog then, at a minimum, the prolog must state the version of the XML specification to which the document conforms. At the time of this writing, the latest version of the XML specification is 1.0. As discussed in a later section, documents may also use the prolog to specify the character set used to encode the document.

XML documents are composed of *elements*. Each document must have a *root element*. It must have one and only one root element. If you are familiar with tree structures, the root element is the lone node at the top of the tree (or rather, the inverted tree) from which all other nodes in the tree descend. The sample document is an invoice for goods shipped to a customer. Since the content of an XML document is supposed to be self-describing, we add a root element having the name “invoice”:

```
<?xml version="1.0"?>
<invoice>
</invoice>
```

Notice that the document contains two lines for the root element. The first line contains the start tag for “invoice.” A very simple start tag begins with the character ‘<’ followed by the element name and ends with the character ‘>’. XML documents must have balanced tags, meaning that each start tag must have an end tag. An end tag, as shown on the third line of the document, begins with the characters ‘</’ followed by the element name and ends with the character ‘>’. Each element in the document must have a start tag and an end tag.

Right now, the invoice element is empty. If the element were to remain empty in the finished document then the start and end tag can be shortened into an *empty element tag* as follows:

```
<?xml version="1.0"?>
<invoice/>
```

Empty element tags may be used for any element that has no content. Usually, an element does have content. An element's content is anything that exists between the element's start tag and its end tag. An element may have text or other elements within its content. Recalling the tree analogy, the root element of the document may contain child elements. The child elements may also contain child elements, and so forth.

For this example, the invoice needs to indicate the payee and what was purchased. In this example, the elements are indented to improve the readability of the document. The XML specification does not require you to indent the content of your documents:

```
<?xml version="1.0"?>
<invoice>
  <payee>
  Wile E. Coyote
    <company>
    Super Genius Enterprises
    </company>
  </payee>
  <item quantity="10" price="19.99">
  dynamite sticks, gross
  </item>
</invoice>
```

The "payee" element identifies the person paying for the merchandise. It contains a set of text "Wile E. Coyote" and a child element identifying the payee's company. This is referred to as *mixed content* because it contains both textual data and a child element. The "company" element has *simple content* because it contains only text. If an element contains only elements then it is said to have *element content*.

The "invoice" element also contains an "item" element identifying the merchandise purchased by Mr. Coyote. Notice that the "item" element has two *attributes*, "quantity" and "price." Attributes in XML are similar to attributes in HTML. They contain information that pertains to an element. The value assigned to an attribute must be enclosed by single quotes or double quotes.

The quantity and price attributes could very well have been written as child elements of item, as shown in the following example:

```
<?xml version="1.0"?>
<invoice>
  <payee>
  Wile E. Coyote
    <company>
    Super Genius Enterprises
    </company>
  </payee>
  <item>
  dynamite sticks, gross
    <quantity>
    10
    </quantity>
    <price>
    19.99
    </price>
  </item>
</invoice>
```

There are various ideas as to what should be an attribute and what should be an element. Sometimes it comes down to personal preference. Sometimes it comes down to how the data is to be used. If the data describes something about the element but is not of general interest to the application processing the document then it should most likely be an attribute.

To finish the example, the invoice needs information about the sales tax and total amount due:

```
<?xml version="1.0"?>
<invoice>
  <payee>
  Wile E. Coyote
    <company>
    Super Genius Enterprises
    </company>
  </payee>
  <item quantity="10" price="19.99">
  dynamite sticks, gross
  </item>
  <tax>
  12.00
  </tax>
  <totaldue>
  211.90
  </totaldue>
</invoice>
```

Now look at the XML document that was created. It is easy to see the document contains an invoice. It is easy to determine the identity of the payee, what they purchased, and how much they owe. The content of the document is self-describing. However, the document does not state how it is to be displayed. That job is left to an external software application, such as a job that prints invoices on a nightly basis.

The XML document in this example is considered to be *well-formed*. The XML specification states that an XML document is well-formed only if it meets the syntax requirements listed in the XML specification. In this particular example, the important requirements are as follows:

- The document must have only one root element.
- Each start tag has an end tag.
- The end tags are properly nested. In other words, a parent's end tag does not occur before any of its children's end tags.
- The element names are valid.

Character sets

The XML specification supports a wide variety of character sets. The XML specification requires that XML processors be able to read documents encoded in either UTF-8 or UTF-16. Documents encoded in UTF-16 must begin with a Byte Order Mark that indicates whether the encoding is UTF-16 big endian or UTF-16 little endian. The specification requires that any documents encoded in a format other than UTF-8 or UTF-16 place an XML encoding declaration at the beginning of the document.

The encoding declaration occurs within the document's prolog, as shown in the following example:

```
<?xml version="1.0" encoding="EUC-JP"?>
```

An XML processor is not required to recognize any encodings other than UTF-8 and UTF-16. The XMLPartner parser recognizes UTF-8 and UTF-16.

Case-sensitivity and white space

Element tags and attribute names within XML documents are case-sensitive. The element name "invoice" does not match the element name "Invoice." This means that when you define the element names for a document, you should decide upon a particular style and stick with it.

White space includes special characters such as spaces, tabs, and line terminators. By default, XML processors preserve all white space within a document. This is because the white space may be pertinent to the content.

There are two exceptions to this rule. First, text files written by applications running in the Windows operating systems tend to terminate lines with carriage return and line feed characters. An XML processor transforms line terminators into a single line feed character. Second, white space between element tags is considered extraneous and is removed.

Parsed vs. unparsed data

Since an XML element may contain child elements, all text within an element is parsed. This means that if your text needs to include a character that XML uses to delimit element tags then XML will treat those characters as tag delimiters. For example, the following text would be parsed as a text section followed by a mal-formed element start tag:

```
Thank you for your order. Sincerely <your name here>
```

When using the '<' character in text, it may be represented using "&#3C;" or "<". The ampersand character may be represented using "&#26;" or "&". The right angle bracket (>) may be represented using "&#3E;" or ">". Given these rules, the previous line of text may be rewritten as follows:

```
Thank you for your order. Sincerely &lt;your name here&gt;
```

Note that the apostrophe or single-quote character (') may be represented as "'" and the double-quote character (") as """.

If your document has data that contains many right and left angle brackets or an XML processor should never parse the data, place the data within a CDATA section. Everything placed inside a CDATA section is ignored by the XML processor. A CDATA section begins with the tag "<![CDATA[" and ends with the tag "]]>".

The following document contains a set of HTML markup within a CDATA section:

```
<?xml version="1.0"?>
<memo>
  <to>
   Customer Service representatives
  </to>
  <subject>
   Common courtesy
  </subject>
  <body>
  <![CDATA[
    When talking with customers on the telephone,
    please <b>DO NOT BELCH!!</b>. Thank you.
    ]]>
  </body>
</memo>
```

You may also embed comments within an XML document. Comments start with the tag "<!--" and end with the tag "-->". An XML processor ignores all text between the start and end tags of a comment section. Following is a version of the memo containing a comment section:


```
<?xml version="1.0"?>
<!--Generated 31.01.2001-->
<memo>
  <to>
   Customer Service representatives
  </to>
  <subject>
   Common courtesy
  </subject>
  <body>
  <![CDATA[
    When talking with customers on the telephone,
    please <b>DO NOT BELCH!!</b>. Thank you.
    ]]>
  </body>
</memo>
```

Document Type Definitions

The XML document created earlier in this chapter does a good job of describing what it contains. But how do you know the data is valid? What if the application printing the invoice expected to find a "customer" element instead of a "payee" element?

XML allows a document to use a Document Type Definition (DTD) to define the possible content of an XML document. The DTD may be internal to the document or it may be external to the document. The DTD defines what elements may be used in the document, the content of the elements, and the relationship of the elements to one another. It also defines the attributes that may be specified for any given element.

DTDs travel with the document. Applications expecting to receive XML documents in a certain format can use the DTD to verify the format of incoming documents. People within special fields of interest can use a DTD to define a consistent format for information pertaining to their field.

While all XML processors are required to detect documents that are not well-formed, XML processors are not required to validate a document against its DTD. A non-validating processor may read the DTD but it will not verify the contents of the document meet the requirements stated in its DTD. A validating processor verifies that a document meets the

requirements stated in its DTD. XMLPartner is a non-validating processor. However, XMLPartner provides you with the ability to verify a document against its DTD using the XMLPartner Document Object Model (DOM).

The remainder of this section shows you how to create a DTD for the invoice example used earlier in this chapter. The first step is to declare the DTD:

```
<!DOCTYPE invoice []>
```

A DTD begins with the tag "<!DOCTYPE". The first piece of information within the DTD is the name of the root element. This happens to be "invoice" in this example. The brackets after the name of the root element are to contain the element and attribute declarations for the document. An element declaration states what elements the document may contain. An attribute declaration states the attributes that may be associated with a particular element, the data type of the attributes, and default values, if any, for each attribute.

In the invoice example, we need to declare the "payee", "item", "tax", and "totaldue" elements. Adding these to the DTD results in the following:

```
<!DOCTYPE invoice [
  <!ELEMENT invoice (payee, item+, tax, totaldue)>
  <!ELEMENT payee (#PCDATA | company)>
  <!ELEMENT company (#PCDATA)>
  <!ELEMENT item (#PCDATA)>
  <!ELEMENT tax (#PCDATA)>
  <!ELEMENT totaldue (#PCDATA)>
]>
```

An element declaration starts with the tag '<!ELEMENT' and ends with the tag '>'. The first item in the element declaration is the name of the element. Remember that element names are case-sensitive in XML. Immediately following the element name is the content declaration. The content declaration may be the string "ANY", "EMPTY", a list of elements, or a mixture of PCDATA and elements. ANY means that any text or elements may occur within the element as long as the information is well-formed. EMPTY means that the element contains nothing whatsoever. PCDATA means parsed text that does not contain any elements.

In this example, the "payee" element contains some text (i.e., the payee's name) followed by a "company" element. The "company" element contains some text, as do the remaining elements in the document.

The last set of information to declare for the invoice is the attributes associated with the "item" element:

```
<!DOCTYPE invoice [
  <!ELEMENT invoice (payee, item+, tax, totaldue)>
  <!ELEMENT payee (#PCDATA | company)>
  <!ELEMENT company (#PCDATA)>
  <!ELEMENT item (#PCDATA)>
  <!ATTLIST item
      quantity CDATA #REQUIRED
      price    CDATA #REQUIRED
  >
  <!ELEMENT tax (#PCDATA)>
  <!ELEMENT totaldue (#PCDATA)>
]>
```

An attribute list declaration begins with the tag '<!ATTLIST' and ends with the tag '>'. The first piece of information within the declaration is the name of the element to which the declaration applies. A list of individual attribute declarations follows the element name. In this example, we declare the attributes "quantity" and "price." Both are assigned the type CDATA and both are marked as #REQUIRED meaning they must appear in every instance of the "item" element.

The element and attribute list declarations created for the invoice document illustrate only a few of the features available in DTDs. For example, a DTD allows you to indicate which elements are required and which elements are optional. A DTD allows you to assign default values to an attribute and to declare attributes of various types. Please read the XML specification or an introductory XML book for more information about element and attribute-list declarations.

Once you have created a DTD, you may embed the DTD itself or a reference to the DTD within the document. If the DTD is to be frequently used by a number of XML documents, it is usually best to place the DTD in a place accessible by all of the documents such as a network directory, a Web site, or an FTP site.

The XML document now looks like this after adding the DTD:

```
<?xml version="1.0"?>
<!--The DTD starts here -->
<!DOCTYPE invoice [
  <!ELEMENT invoice (payee, item+, tax, totaldue)>
  <!ELEMENT payee (#PCDATA | company)>
  <!ELEMENT company (#PCDATA)>
  <!ELEMENT item (#PCDATA)>
  <!ATTLIST item
      quantity CDATA #REQUIRED
      price    CDATA #REQUIRED
  >
  <!ELEMENT tax (#PCDATA)>
  <!ELEMENT totaldue (#PCDATA)>
]>
<!--The document starts here-->
<invoice>
  <payee>
  Wile E. Coyote
    <company>
    Super Genius Enterprises
    </company>
  </payee>
  <item quantity="10" price="19.99">
  dynamite sticks, gross
  </item>
  <tax>
  12.00
  </tax>
  <totaldue>
  211.90
  </totaldue>
</invoice>
```

To make the DTD available to all software applications that use an invoice document, place the DTD on an intranet at http://internal.acme.com/xml/dtds/invoice.dtd. This is referred to as an external DTD. The document now looks like the following:


```
<?xml version="1.0"?>
<!-- Following is the reference to the external DTD -->
<!DOCTYPE invoice SYSTEM
  "http://internal.acme.com/xml/dtds/invoice.dtd"]>
<!--The document starts here-->
<invoice>
  <payee>
  Wile E. Coyote
    <company>
    Super Genius Enterprises
    </company>
  </payee>
  <item quantity="10" price="19.99">
  dynamite sticks, gross
  </item>
  <tax>
  12.00
  </tax>
  <totaldue>
  211.90
  </totaldue>
</invoice>
```

The SYSTEM keyword within the DTD declaration tells the XML processor that the DTD is stored externally. The string after the SYSTEM keyword contains an absolute or relative path to the DTD.

XMLPartner recognizes both internal and external DTDs. When using the XMLPartner Document Object Model (DOM), you can construct DTDs dynamically and save them as external or internal DTDs.

Entities

The term *entity* refers to a unit of storage. An XML document is an entity. The external DTD shown in the previous section is an entity. A set of replacement text within an XML document is also referred to as an entity. This section shows you how to take advantage of entities within an XML document.

Entities may be categorized as *general* entities or *parameter* entities. A general entity is usable within any XML document while a parameter entity may be used only in DTDs. This section focuses on general entities.

Entities are declared in a document's DTD. For example, the following document declares an entity for a copyright:

```
<!DOCTYPE letter [
  <!ENTITY CopyRight "Copyright 2001, TurboPower Software">
  ...
]>
```

When the XML processor encounters a reference to the entity, the processor replaces the reference with the text from the entity declaration. An entity reference is comprised of an ampersand character (&), the name of the entity, and a semi-colon (;). An example of a reference to the CopyRight entity is as follows:

```
<titlepage>
XMLPartner
&CopyRight;
</titlepage>
```

When parsed, the processor expands the entity references and presents the document to the application as follows:

```
<titlepage>
XMLPartner
Copyright 2001, TurboPower Software
</titlepage>
```

When the processor expands the entity reference, it replaces the reference with the text and then parses the replacement text. This means that you can embed entity references within the replacement text of an entity, as shown in the following example:

```
<!ENTITY UpdateDate "1/31/2001">
<!ENTITY LastUpdate "This Web page last updated &UpdateDate;">
```

An entity may refer to a file that is external to the XML document. When the processor sees a reference to that entity, it replaces the entity reference with the contents of the file and continues parsing the contents. Thus, you could build an XML document from other XML documents. In the following example, an XML document builds a complete specification using several external entities:


```
<?xml version="1.0"?>
<!DOCTYPE spec [
  <!ENTITY titlePage SYSTEM "..\specs\milstd1672\title.xml">
  <!ENTITY scope SYSTEM "..\prod\capstone\scope.xml">
  <!ENTITY section1 SYSTEM "..\prod\capstone\section1.xml">
  <!ENTITY section2 SYSTEM "..\prod\capstone\section2.xml">
  <!ENTITY section3 SYSTEM "..\prod\capstone\section3.xml">
 ]>
<spec>
&titlePage
&scope
&section1
&section2
&section3
</spec>
```

A reference to an external entity requires that the SYSTEM keyword and an absolute or relative path to a file follow the entity name. XMLPartner recognizes external entities. The entity may be located on a local or network hard drive, a Web site, or an FTP site. It is entirely possible to create a locally stored document that references XML documents available from a number of different Web sites.

This section introduced you to the concept of internal and external entities. Please see the XML specification or an introductory XML book for more information about other entity options and types.

Using XML

XML is a popular buzzword, but how do you use it? What is it good for? The purpose of this section is to give you ideas for using XML. This is not a comprehensive overview of the possibilities. Instead, view it as something to prime the pump.

Save configuration data

In the Windows world, many software applications save user settings and other information within an INI file. INI files are plain text. Any one can read and, assuming the information is worded properly, can understand and modify the contents of an INI file. However, INI files are limited to two dimensions when structuring the data. The first dimension is a section (e.g., WindowSettings, History) and the second dimension is a data value.

Using XML to store configuration data offers the same advantages of INI files with the benefit of providing an unlimited number of dimensions for nesting and structuring the data. For example, the following document stores an application's window position and most recently used file history:

```
<softapp>
  <mainwindow>
    <state size="normal" x="100" y="100" ht="300" wid="400">
    <MRUFile>
      <file>c:\bloat.txt</file>
      <file>d:\clients\wile\memo20010208.txt</file>
      <file>d:\rec\castlerock.txt</file>
      <file>c:\autoexec.bat</file>
    </MRUFile>
  </mainwindow>
</softapp>
```

An existing, real-world application uses XML documents to build a graphical interface that collects information from its users. If the creators of the application decide they need to change existing questions or add new questions, they merely change the XML file.

Several development tool vendors save their information in XML format. In the Linux world, Glade is a user interface builder that outputs interface description files in XML. In the first four versions of Borland's Delphi, all forms were saved in a binary format. Beginning with version 5.0, Delphi now saves its files in an XML format. Saving information in XML increases the likelihood that the information may cross platform boundaries.

Temporary data storage

XML is a good way to temporarily store data. For example, an application used by traveling salesman is to collect data from the salesman. The data is transferred to the corporate database once it has been finalized by the salesman and approved by their manager. If the salesman does not have a connection to the corporate database while on the road, the application could save the information in XML format. An import program could then transfer the data into the corporate database at a later time.

Please note that XML is not designed to be a database. It is not designed to support high-speed searches or multi-way joins between various sets of data. However, it can be useful for storing information that is not yet ready for inclusion in a database.

Electronic exchange of information

XML is ideal for the exchange of purchasing requests, invoices, letters, and any other type of business document. For example, the accounting, inventory, purchasing, and order entry applications used by a company are hosted on various hardware platforms. An AS/400 runs accounting and inventory. The purchasing and order entry applications are hosted by a database on a Windows NT box. The Web site is managed by a Linux server and is another avenue for collecting orders from customers. How are all these systems to exchange the documents necessary to run the company, such as invoices, purchase requests, and orders?

Given the right software, each system is entirely capable of reading and writing XML documents. When an order is received through the Web site, the Web order entry application creates an XML document and sends it to the main order entry system. The order entry system submits a shipping request in XML format to the inventory system. When the customer's order has shipped, an invoice request in XML format is forwarded to the accounting system.

Businesses may also exchange XML documents among one another. In the previous scenario, if the buy guides generated by a purchasing agent indicates the need to buy more inventory, the purchasing agent generates purchasing requests for one or more vendors. The purchasing requests are transmitted to the vendor order entry systems using XML documents.

Note that in this situation, the XML documents have to be in well-defined formats. Using XML means that you must spend time upfront deciding what data is to go into a document and how it is to be organized.

Communication between business objects

If you are developing a multi-tier application that implements layers of business objects, it may be viable for the business objects to communicate with the front end user interfaces and between each other using XML. This approach is very useful for situations where the business objects are talking to different database engines. For example a set of accounting business objects that interacts with a Microsoft SQL Server database may need to communicate with order entry business objects that interface with an Oracle database.

Share data among geographically diverse groups

People sharing a common interest could store information within XML documents. If the people are spread around the world, the documents could be published on the Web or a private FTP site. The probability of everyone having the same operating system and software is low, so storing the information in XML files means that any person can view the information as long as they have a text editor, Web browser, or some other tool that processes XML documents.

Create new languages

Because XML is a *metalanguage* (i.e., a language about a language), you have the freedom to create new languages for specific fields of interest or areas of business. For example, Mathematical Markup Language (MathML) is useful for representing mathematical expressions. See http://www.w3.org/Math/ for information about MathML. The Chemical Markup Language (CML) is an XML vocabulary for representing and drawing molecular information. For more information about CML, see http://www.xml-cml.org/.

Drive a Web site's content

The way in which a Web site's content is generated ranges from the manual typing of information into a text editor to automated generation via server-side scripts or applications. An alternative is to use XML as a source of content. With the use of Cascading Style Sheets (CSS) or Extensible Style Sheet Language for Transformations (XSLT), XML documents may be converted into HTML files. In addition, Microsoft provides an XML data source that allows you to bind an XML document to an HTML table.

The Advantages of XMLPartner

Since XML is a format for defining and storing information, you cannot execute or run an XML document. Instead, you need something that helps your applications read and write XML documents. XMLPartner provides you with two different ways to retrieve information from XML documents, a way to query information in a document, and a way to write new or changed documents.

Event-driven parsing

Event-driven parsing is best suited for applications needing to make a one-time, read-only pass through the document. The TXpParser class, detailed in Chapter 3, provides an event-driven model for parsing XML documents. It can parse documents stored in-memory, on local or network drives, or on the Internet. As it parses a document, it raises events. Your application receives events for which it has declared handlers. For example, if your application needs to find all occurrences of an “Invoice” element then it provides a handler for the OnStartElement event. Each time the parser encounters an element, it calls the handler for OnStartElement and identifies the name of the element encountered.

Document Object Model

The Document Object Model (DOM) is best suited for applications needing to create, manipulate, change, or query an XML document. The TXpObjModel class, detailed in Chapter 4 on page 81, provides a way to build an *object model* from an XML document or to create a new XML document using an object model. The term object model means that the XML document is read and organized into a structure of objects. For example, the invoice example used on page 27 earlier in this chapter would be structured into an object model as in Figure 2.1:

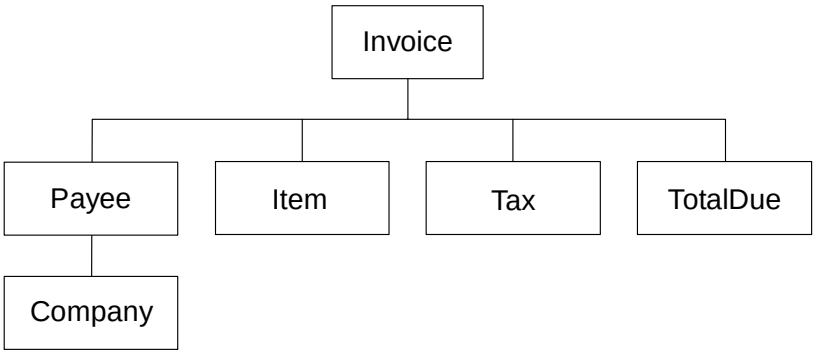


Figure 2.1: The object model for the invoice example

The TXpObjModel reads the entire XML document, and any external XML documents referenced by the root document, into an in-memory object structure. Your applications can then do such things as use XML Path Language (XPath) to search for information within the document, navigate through the document, rearrange the document, change attributes or add new attributes to elements, and save the changed document. The XMLPartner DOM interface also provides additional classes to support the reading, modification, and creation of Document Type Definitions (DTD). This means that your applications can use the DOM to validate a document against its DTD and create new DTDs from scratch.

Chapter 3: Event-Driven Parsing

TXpParser is TurboPower's event-driven component for parsing XML documents. This component serves as an interface between an XML document and an application. As the document is parsed, the component raises events as it encounters the document's structures. Applications using this component need to assign event handlers to respond to these events.

TXpParser is a non-validating parser. This means it can parse XML documents that meet the XML specification's well-formedness constraints. It does not validate the constraints expressed in the document type definition.

The TXpParser component doesn't maintain the document or its structures after it's parsed. This means it can't be used to modify, create, or search an XML document. Use TXpObjModel for these tasks. See "TXpObjModel Component" on page 81 for details.

Character Sets

TXpParser recognizes UTF-16 big endian, UTF-16 little endian, UTF-8, and ASCII character sets. The character set is determined by first looking for a byte order mark (BOM) when the XML document is opened. The BOM is a signature used to differentiate between little endian and big endian character encoding. If the BOM is present, the character set is determined to be UTF-16; the BOM then is read to further determine if the document is encoded as big endian or little endian. If the BOM isn't present, the character set is determined to be UTF-8 or ASCII, which is a subset of UTF-8.

If the encoding declaration in the document's prolog contains an unsupported character set, such as EUC-JP, TXpParser raises an exception.

Using This Component

For an example of how to use the TXpParser, suppose there is an XML document standard that represents text with the font information saved as attributes and the style (bold, italic, and underlined) information saved as elements of a particular name. TXpParser can be used to create a viewer for these types of documents. The rest of this section describes how to use TXpParser to create a viewer to display an XML document of this type.

To those familiar with XML, this may appear to be an improper use of XML. XML is normally used to organize data while HTML is normally used to display data. From one perspective, we are organizing font and formatting information. Although HTML is better suited to displaying data, this example demonstrates how to use the TXpPartner component.

Below is an example of the XML documents our viewer can display:

```
<Contents>
   <P>
      <Text FontName="Arial" PointSize="10" Color="0">
         The importance of properly formatting data
      </Text>
      <Text FontName="Arial" PointSize="10" Color="0">
         <B xml:space='preserve'> can not </B>
      </Text>
      <Text FontName="Modern" PointSize="20" Color="255">
         be over emphasized.
      </Text>
      <Text FontName="Arial" PointSize="10" Color="0">
        <I xml:space='preserve'>
This line is italicized.</I>
      </Text>
      <Text FontName="Arial" PointSize="10" Color="0">
         <I>
            <U xml:space='preserve'>
This line is underlined and italicized.</U>
         </I>
      </Text>
   </P>
</Contents>
```

Designing an interface

The complete project is called TextViewer and is located in the Examples subdirectory (see Figure 3.1).

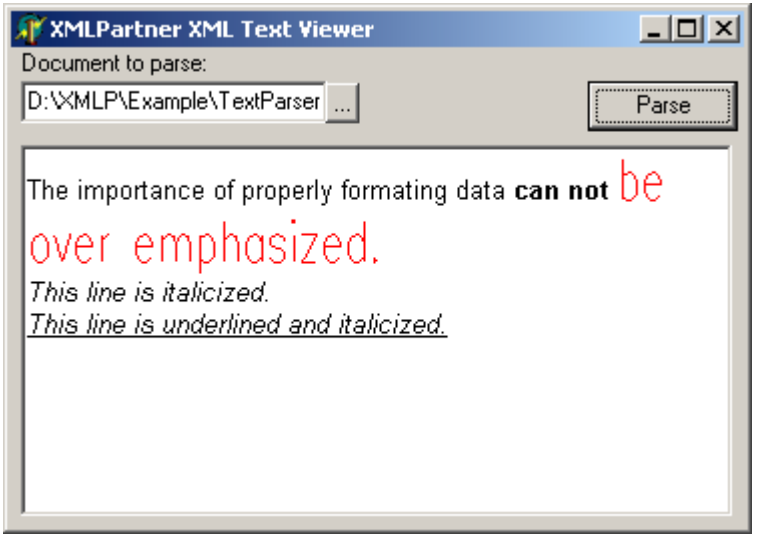


Figure 3.1: XMLPartner interface

The interface doesn't need to be very complicated. It needs a way to select a file (XML document) to parse, a way to initiate parsing, and a way to view the parsed documents or results. The only specific requirement of this example is a rich edit component to display the formatted data. After setting up the interface, you must add the TXpComponent and establish the event handlers.

Establishing the event handlers

Familiarization with TXpParser's events and their order is essential. This example requires events to know when the predefined elements and attributes are being processed, what character data is in the predefined elements, and when to turn off the style settings. The application uses the OnAttribute, OnStartElement, OnEndElement, and OnCharData events to serve these needs.

Setting the font

The font used to display the text is set using predefined attributes. An attribute is a named piece of information in an element's start tag or empty-element tag. The FontName attribute designates which font to use when displaying the text. The PointSize attribute is the size of the font in points. The Color attribute is the color to use when displaying text.

The OnAttribute event is the first one raised in this example and allows the viewer application to catch the predefined attributes and set the font appropriately. This example checks for the predefined attribute names (sName) and, if found assigns its value (sValue) to the appropriate font property. The event handler follows:

```
procedure TfrmMain.ParserAttribute(
  oOwner : TObject; sName, sValue : DOMString;
  bSpecified : Boolean);
begin
  if sName = 'FontName' then
    Memo.SelAttributes.Name := sValue
  else if sName = 'PointSize' then
    Memo.SelAttributes.Size := StrToInt(sValue)
  else if sName = 'Color' then
    Memo.SelAttributes.Color := StrToInt(sValue);
end;
```

Setting the text's style

Bold, italics, and underlined text is designated by predefined element tags. Element tags are the markings used to identify the beginning and end of individual segments of data. <B> and </B> tags surround bold text, <I> and </I> surround italicized text, and <U> and </U> surround underlined text.

OnStartElement is raised for each start tag found (after raising the OnAttribute event for each attribute). A start tag marks the beginning of an element. <B> is the start tag used to identify bold text in the example XML document. As shown in the following event handler for OnStartElement, the example viewer uses this event to check for the predefined element tags and set the font's style appropriately:


```
procedure TfrmMain.ParserStartElement(
  oOwner : TObject; sValue : DOMString);
begin
  if sValue = 'B' then
    Memo.SelAttributes.Style :=
      Memo.SelAttributes.Style + [fsBold];
  if sValue = 'I' then
    Memo.SelAttributes.Style :=
      Memo.SelAttributes.Style + [fsItalic];
  if sValue = 'U' then
    Memo.SelAttributes.Style := Memo.SelAttributes.Style +
[fsUnderline];
end;
```

When the OnStartElement event is raised, the element's name (sValue) is compared against the predefined names. If found, the font's style is set using the rich edit's SelAttributes property. These property settings apply to the text added to the rich edit component using the SelText property.

Retrieving the text to display

OnCharData is raised any time character data is found. Character data is the text between the start and end tags. The text in the sample XML document is stored as character data. The following event handler demonstrates how text is retrieved from OnCharData event and placed in the rich edit box:

```
procedure TfrmMain.ParserCharData(
  oOwner : TObject; sValue : DOMString);
begin
  Memo.SelLength := 0;
  Memo.SelText := sValue;
end;
```

Since the OnAttribute and OnStartElement events are raised before the OnCharacter data, the appropriate font and styles have been applied and the text is added with the correct font and styles.

Turning off style settings

OnEndElement is raised after an element is completely parsed. The example viewer uses this event to check for completion of the predefined elements and turn off the related style setting if needed. For example, the OnEndElement's sValue is 'B' after parsing a section of bold text so it turns off the font's bold setting. The OnEndElement event handler follows:

```
procedure TfrmMain.ParserEndElement(
  oOwner : TObject; sValue : DOMString);
begin
  if sValue = 'B' then
    Memo.SelAttributes.Style :=
      Memo.SelAttributes.Style - [fsBold];
  if sValue = 'I' then
    Memo.SelAttributes.Style :=
      Memo.SelAttributes.Style - [fsItalic];
  if sValue = 'U' then
    Memo.SelAttributes.Style :=
      Memo.SelAttributes.Style - [fsUnderline];
end;
```

Initiating the parse

The last step is to initiate the parse. After selecting an XML document in the prescribed format, initiate the parse by calling ParseDataSource. The following example demonstrates initiating the parse and checking for errors by clicking a button:

```
procedure TfrmMain.btnParseClick(Sender : TObject);
begin
  Memo.Lines.Clear;
  if edtFile.Text <> '' then begin
    if not Parser.ParseDataSource(edtFile.Text) then
      Memo.Lines.Assign(Parser.Errors);
  end else
    ShowMessage('You must select an XML document to parse.');
end;
```

The complete project is called TextViewer and is located in the Examples subdirectory.

Handling Errors

TXpParser raises TXpParserError exceptions when deviations from the XML specification's for being well-formed are encountered. These errors are maintained in a list that is accessed using the Errors property or the GetErrorMsg method. Use the ErrorCount property to find out how many errors, if any, were encountered during the parse. Since TXpParser is not a validating parser, errors are not raised because of deviations from the document type definition (DTD).

Users have a choice of letting the TXpParser catch these exceptions or raising them to the application. By default TXpParser catches the exceptions. Set the RaiseErrors property to True to raise them to the application.

Memory Usage

TXpParser is able to parse very large XML documents because of its strategy for conserving memory usage. The parser buffers XML documents to prevent reading them in all at once. Reading in a large document all at once takes memory away from other applications and could cause memory to be swapped to disk temporarily. Reading and writing memory to and from disk slows down TXpParser and other applications running on your system.

Buffer size is determined by the BufferSize property. The default buffer size is 8,192 bytes. The default size is the optimum size based on tests reading (not parsing) XML documents with various size buffers, documents, and character sets. Table 3.1 summarizes how many milliseconds are required to read a 3.3MB XML document with various size buffers.

Table 3.2: *Sample read times*

Buffer Size	ANSI	UTF-8	UTF-16	UTF-16LE
4,096	6820	6930	9323	9374
8,192	6680	6669	9284	9033
65,536	7842	7831	10896	10705
524,288	7962	7951	11026	10825
1,048,576	8492	8473	11776	11597

The second part of the memory strategy keeps external entities separate from the XML document. An external entity is a part of the XML document that is not stored in the document. Each external entity is parsed within its own buffer. This prevents the main document buffer from expanding and also reduces the possibility of memory being swapped to disk.

TXpParser Component

Hierarchy

TComponent (VCL)

3

TXpParser (XpParser)

Properties

BufferSize
ErrorCount
Errors
InCharSet
NormalizeData
Password
RaiseErrors
UserName

Methods

GetErrorMsg
ParseDataSource
ParseMemory

Events

OnAttribute
OnCDATASection
OnCharData
OnComment
OnDocTypeDecl
OnDTDAttDefinition
OnDTDAttlistEnd
OnDTDAttlistStart
OnDTDConditionalEnd
OnDTDConditionalStart
OnEndDocument
OnEndElement
OnDTDElementContent
OnDTDElementEnd
OnDTDElementGroupEnd
OnDTDElementGroupStart
OnDTDElementStart
OnDTDEnd
OnDTDEntityExternal
OnDTDEntityInternal
OnDTDExternalEnd
OnDTDExternalStart
OnDTDNotation
OnDTDStart
OnNonXMLEntity
OnPreserveSpace
OnProcessingInstruction
OnResolveEntity
OnStartDocument
OnStartElement

Reference Section

BufferSize — property

```
property BufferSize : Integer
```

Default: 8192

Determines the size (in bytes) of the buffer used to parse the source XML document.

TXpParser uses a buffer to prevent placing a whole XML document in memory at once. See the Memory usage section earlier in the chapter for more information.

ErrorCount — read-only property

```
property ErrorCount : Integer
```

Returns the number of errors found during the most recent call to ParseDataSource or ParseMemory.

If ErrorCount is 0, the parsed document meets the XML specification's well-formedness constraints and was successfully parsed. If ErrorCount is not 0, retrieve the complete list of error messages using the Errors property or each of the individual messages using the GetErrorMsg method.

The following example shows one way to use the ErrorCount property:

```
MyMemo.Lines.BeginUpdate;
try
  if Parser.ErrorCount > 0 then
    MyMemo.Lines.Assign(Parser.Errors)
  else
    MyMemo.Lines.Add('No errors found');
finally
  MyMemo.Lines.EndUpdate;
end;
```

See also: Errors, GetErrorMsg, RaiseErrors

Errors — read-only property

```
property Errors : TStringList
```

Returns a list of the error messages received during the previous call to ParseDataSource or ParseMemory.

If a document doesn't meet the XML specification's well-formedness constraints and wasn't parsed successfully, retrieve the complete list of error messages using this property. To access the errors messages individually, use the GetErrorMsg method.

The following example shows how to view error messages in a memo:

```
MyMemo.Lines.BeginUpdate;
try
  if Parser.ErrorCount > 0 then
    MyMemo.Lines.Assign(Parser.Errors)
  else
    MyMemo.Lines.Add('No errors found');
finally
  MyMemo.Lines.EndUpdate;
end;
```

See also: ErrorCount, GetErrorMsg, RaiseErrors

GetErrorMsg — method

```
function GetErrorMsg(const wIdx : Integer) : DOMString;
```

Returns one of the error messages, if any, generated during the previous call to ParseDataSource or ParseMemory.

Use this method to retrieve a specific error message that was generated while parsing. wIdx is the zero-based index of error messages. To retrieve a complete list of error messages, use the Errors property.

The following example shows how to access specific error messages:

```
var
  ErrMsg : String;
  i : Integer;
begin
  for i := 0 to Pred(XmlParser1.ErrorCount) do begin
    ErrMsg := XmlParser1.GetErrorMsg(i);
    if Pos('WININET', ErrMsg) <> 0 then
      ShowMessage(ErrMsg);
  end;
```

See also: ErrorCount, Errors, RaiseErrors

InCharSet — read-only property

```
property InCharSet : TXpCharEncoding
```

```
TXpCharEncoding = (ceUnknown, ceUTF8, ceUTF16LE, ceUTF16BE);
```

Determines the character set of the previously parsed XML document.

The meanings of the TXpCharEncoding values are shown in the following table:

Value	Meaning
ceUnknown	Document has not been parsed or unsupported character set in encoding declaration.
ceUTF8	UTF-8 character set or one of its subsets, such as ASCII.
ceUTF16LE	UTF-16 character set with little endian encoding.
CeUTF16BE	UTF-16 character set with big endian encoding.

See the Character sets section on page 44 for more information.

See also: ParseDataSource, ParseMemory

NormalizeData — property

```
property NormalizeData : Boolean
```

Default: True

Determines if extra white space should be removed from character data.

Set this property to True to strip out all extra white space from character data. Extra white space consists of any leading or trailing white space characters and multiple white space characters found within the character data. Character data remains as found when this property is set to False.

The following example shows the results of normalization:

Before:

```
"     The dog        jumped over        the cat.           "
```

After:

```
"The dog jumped over the cat."
```

Normalized data can cause problems when trying to use leading and trailing spaces to format character data from multiple elements. For example:

```
<Contents>
<P>
<Font FontName="Arial" PointSize="10" Color="#000000">
The importance of properly formatting data </Font>
<Font FontName="Arial" PointSize="10" Color="#000000">
<B>can not </B>
</Font>
<Font FontName="Arial" PointSize="10" Color="#000000">
be over emphasized.
</Font>
</P>
</Contents>
```

Using this XML document, the result of combining character data from the three elements produces:

```
The importance of properly formatting datacan notbe over
emphasized.
```

Note that there isn't a space between "data" and "can" or "not" and "be", although there is spacing to account for combining the strings. The solution is to use the xml:space attribute as follows.

```
<Contents>
<P>
<Font FontName="Arial" PointSize="10" Color="#000000" xml:space =
'preserve'>The importance of properly formatting data </Font>
<Font FontName="Arial" PointSize="10" Color="#000000" xml:space =
'preserve'>
<B>can not </B></Font>
<Font FontName="Arial" PointSize="10" Color="#000000" xml:space =
'default'>be over emphasized.</Font>
</P>
</Contents>
```

Note: This property can be overridden using OnPreserveSpace or the xml:space attribute.

See also: OnPreserveSpace

```
property OnAttribute : TXpAttributeEvent
```

```
TXpAttributeEvent = procedure(
  oOwner : TObject; sName, sValue : DOMString;
  bSpecified : Boolean) of object;
```


Defines an event handler called when an attribute is found in an element start tag.

This event is raised for each attribute in the tag. An attribute is a named piece of information in an element's start tag or empty-element tag. The following element has an attribute named "position" with a value of 11:

```
<record position="11">
     <group>Heart</group>
     <title>Greatest Hits</title>
</record>
```

The OnAttribute events are raised before the OnStartElement event for the element to which they pertain. oOwner is the TXpParser object raising the event. sName is the name of the attribute. sValue is the value of the attribute. bSpecified is True if sValue was specified in the element tag and False if it is the default value.

The following example shows how to use this event to set the font in a memo component:

```
procedure TForm.XmlParser1Attribute(
  oOwner : TObject; sName, sValue : DOMString;
  bSpecified : Boolean);
begin
  if sName = 'FontName' then
    Memo.Font.Name := sValue
  else
    Memo.Font.Name := 'Arial';
end;
```

See also: OnStartElement

OnCDATASection event

```
property OnCDATASection : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

Defines an event handler called when a CDATA section is encountered.

A CDATA section is a means of defining literal data is not parsed. oOwner is the TXpParser object generating the event. sValue is the unparsed data excluding the "<![CDATA[" and "]]>" flags.

See also: OnCharData

OnCharData event

```
property OnCharData : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

Defines an event handler called when character data is found.

This event is raised once for each element containing character data. Character data is any data that is not markup data. In the following example, "John" (without quotes) is CDATA:

```
<Name>John</Name>
```

oOwner is the TXpParser object raising the event. sValue contains the character data.

In the following example, all character data from the current element is added to a memo component:

```
procedure TForm1.XmlParserCharData(
  oOwner : TObject;
sValue : DOMString);
begin
  Memo.Lines.Add(sValue);
end;
```

See also: OnCDATASection

OnComment — event

```
property OnComment : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

Defines an event handler called when a comment is found.

oOwner is the TXpParser object firing the event. sValue contains the comment text.

OnDocTypeDecl — event

```
property OnDocTypeDecl : TXpDocTypeDeclEvent
```

```
TXpDocTypeDeclEvent = procedure(
  oOwner : TObject; sDecl, sId0, sId1 : DOMString) of object;
```

Defines an event handler called when a document type (DOCTYPE) declaration is found.

The DOCTYPE declaration tells the parser where to find a document type definition (DTD). A DTD defines a grammar (i.e., validation rules) for an XML document. oOwner is the TXpParser object raising the event. sDecl is the name of the document element.

The DOCTYPE specification allows for a public and system location for the DTD. A public location is used to specify a shared resource. For instance, you may want to use a public DTD in a multi-user system to ensure all users are using the same DTD. Normally these types of shared resources will have a common name, such as “XYZ’sReportFormat.” The TXpParser user is responsible for locating public resources.

A system location is used to specify a specific file, FTP site, or Web address. A system location can be used as a backup to a public resource or as the sole source when a public resource isn’t used. Valid system locations could include “c:\MyDTDs\ReportFormat.dtd”, “file://c:\MyDTDs\ReportFormat.dtd”, “ftp://ftp.xyz.com/ReportFormat.dtd”, or “http://www.xyz.com/ReportFormat.dtd.”

sId0 is the public identifier and sId1 system identifier. TXpParser parses system DTDs, but not public DTDs.

See also: OnResolveEntity

```
property OnDTDAttDefinition : TXpDTDAttDefinitionEvent
```

```
TXpDTDAttDefinitionEvent = procedure(
  oOwner : TObject; sName : DOMString; wAttrType : Integer;
  oEnumeration : TStringList; wValueType : Integer;
  SValue : DOMString) of object;
```

Defines an event handler called for each attribute in a DTD's ATTLIST declaration.

An ATTLIST (attribute-list) declaration defines the attributes for an element type. An attribute is a named piece of information in an element's start tag or empty-element tag. oOwner is the TXpParser object raising the event. sName is the name of the attribute. wAttrType is the attribute type which is defined as follows:

Value	Constant Representation	Meaning
0	ATTRIBUTE_UNDECLARED	Unknown type.
1	ATTRIBUTE_CDATA	Simple text of Name1 characters.
2	ATTRIBUTE_ID	Value derived from the unique identifier of current element.
3	ATTRIBUTE_IDREF	Links to another element by ID.
4	ATTRIBUTE_IDREFS	Links to multiple elements by ID.
5	ATTRIBUTE_ENTITY	Type comes from a predefined entity.
6	ATTRIBUTE_ENTITIES	List of multiple entities.
7	ATTRIBUTE_NMTOKEN	Simple text limited to NameChar characters.
8	ATTRIBUTE_NMTOKENS	Multiple NMTOKEN values.
9	ATTRIBUTE_ENUMERATED	Values must come from oEnumeration.
10	ATTRIBUTE_NOTATION	Value comes from notation type declared in DTD.

oEnumeration is a string list containing the enumerated values if the wAttrType is equal to ENUMERATED. wValueType specifies the type of the default value as shown in the following table:

Value	Constant Representation	Attribute Optional	Meaning
0	ATTRIBUTE_DEFAULT_UNDECLARED	NA	Unknown default type.
1	ATTRIBUTE_DEFAULT_SPECIFIED	Yes	If the attribute appears, the value must match values given in sValue. If the attribute doesn't appear, the parser supplies the default value.
2	ATTRIBUTE_DEFAULT_IMPLIED	Yes	Attribute value is optional.
3	ATTRIBUTE_DEFAULT_REQUIRED	No	Attribute value is required.
4	ATTRIBUTE_DEFAULT_FIXED	Yes	If the attribute is present, the value must match the given value. If not present, the parser will supply the default value.

sValue contains the value of the attribute if the wValueType is equal to SPECIFIED.

See also: OnAttribute, OnDTDAttlistEnd, OnDTDAttlistStart

OnDTDAttlistEnd — event

```
property OnDTDAttlistEnd : TNotifyEvent
```

Defines an event handler that is called when the ATTLIST declaration has completed parsing.

See also: OnDTDAttDefinition, OnDTDAttlistStart

OnDTDAttlistStart — event

```
property OnDTDAttlistStart : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

Defines an event handler that is called when the parser encounters an ATTLIST declaration in the DTD.

An ATTLIST (attribute-list) declaration defines the attributes for an element type. An attribute is a named piece of information in an element's start tag or empty-element tag. oOwner is the TXpParser object firing the event. sValue is the name of the element containing the ATTLIST.

See also: OnAttribute, OnDTDAttDefinition, OnDTDAttlistEnd

OnDTDConditionalEnd — event

```
property OnDTDConditionalEnd : TNotifyEvent
```

Defines an event handler that is called at the end of a conditional section in a DTD.

See also: OnDTDConditionalStart

OnDTDConditionalStart — event

```
property OnDTDConditionalStart : TXpDTDConditionalStartEvent
```

```
TXpDTDConditionalStartEvent = procedure(
  oOwner : TObject; wType : Integer) of object;
```

Defines an event handler that is called at the beginning of a conditional section in a DTD.

A conditional section is a section that, when used with parameter entities, is only parsed when some condition is met. oOwner is the TXpParser object raising the event. wType is the type of conditional section as described in the following table:

Value	Constant Representation	Meaning
0	CONDITIONAL_INCLUDE	Use for validation.
1	CONDITIONAL_IGNORE	Don't use for validation.

See also: OnDTDConditionalEnd

OnDTDElementContent event

```
property OnDTDElementContent : TXpDTDElementContentEvent
```

```
TXpDTDElementContentEvent = procedure(oOwner : TObject;
  sName : DOMString; wOccurrence : Integer;
  wRelation : Integer) of object;
```

Defines an event handler that is called after parsing a DTD's element content.

The element content specifies how an element may occur in the document and its relation to other elements. oOwner is the TXpParser object raising the event. sName is the name of an element or "#PCDATA". wOccurrence is how the element can occur in the document as described in the following table:

Value	Constant Representation	Meaning
0	OCCURS_REQ_NOREPEAT	One child element is required.
1	OCCURS_OPT_NOREPEAT	Zero or one child elements is allowed.
2	OCCURS_OPT_REPEAT	Zero or more child elements allowed.
3	OCCURS_REQ_REPEAT	One or more child elements allowed.

wRelation shows how the elements are related in the document as shown in the following table:

Value	Constant Representation	Meaning
0	REL_OR	Any one of the given (sName) elements must appear.
1	REL_AND	Sequence of elements must appear in given order.

See also: OnDTDElementStart

OnDTDElementEnd event

```
property OnDTDElementEnd : TXpDTDElementEndEvent
```

```
TXpDTDElementEndEvent = procedure(
  oOwner : TObject; sName : DOMString) of object;
```

Defines an event handler that is called after parsing a DTD's ELEMENT declaration.

oOwner is the TXpParser object raising the event. sName is the name of the parsed element.

See also: OnDTDElementStart

```
property OnDTDElementGroupEnd : TXpDTDElementGroupEvent
```

```
TXpDTDElementGroupEvent = procedure(
  oOwner : TObject; wOccurrence, wRelation : Integer) of object;
```


Defines an event handler that is called for a DTD element containing child elements.

An element group is the list of child elements owned by an element of a DTD. oOwner is the TXpParser object raising the event. wOccurences specifies how many child elements are required and allowed as shown in the following table:

Value	Constant Representation	Meaning
0	OCCURS_REQ_NOREPEAT	One child element is required.
1	OCCURS_OPT_NOREPEAT	Zero or one child elements is allowed.
2	OCCURS_OPT_REPEAT	Zero or more child elements allowed.
3	OCCURS_REQ_REPEAT	One or more child elements allowed.

wRelation specifies if all the child elements are required or if only one of the elements is allowed;

Value	Constant Representation	Meaning
0	REL_OR	Any one of the given (sName) elements must appear. This relation is represented in the element type declaration by separating the child elements with a vertical bar.
1	REL_AND	Sequence of elements must appear in given order. This relation is represented in the element type declaration by separating the child elements with a comma.

See also: OnDTDElementGroupStart, OnDTDElementStart

OnDTDElementGroupStart — event

```
property OnDTDElementGroupStart : TNotifyEvent
```

Defines an event handler that is called before parsing a DTD element containing child elements or mixed content.

An element group is an element with mixed content. Mixed content is a combination of character data and child elements. This event is not raised for elements containing only character data.

See also: OnDTDElementGroupEnd

OnDTDElementStart — event

```
property OnDTDElementStart : TXpDTDElementStartEvent

TXpDTDElementStartEvent = procedure(oOwner : TObject;
  sName : DOMString; wContentType : Integer) of object;
```

Defines an event handler that is called after parsing an ELEMENT declaration in a DTD.

A DTD's ELEMENT declaration declares the name of an element type and its allowable child elements. Depending on the content type, this event may be followed by calls to the OnDTDElementGroupEnd, OnDTDElementGroupStart, and OnDTDElementContent events. oOwner is the TXpParser object raising the event. sName is the name of the declared element. wContentType is the content type of the element as described in the following table:

Value	Constant Representation	Meaning
0	CONTENT_UNDECLARED	Unknown content type.
1	CONTENT_ANY	Contains any well-formed content data.
2	CONTENT_EMPTY	Does not contain any text or child elements.
3	CONTENT_MIXED	Contains character data and/or child elements.
4	CONTENT_ELEMENTS	Contains only child elements.

See also: OnDTDElementContent, OnDTDElementGroupEnd, OnDTDElementGroupStart

OnDTDEnd — event

```
property OnDTDEnd : TNotifyEvent
```

Defines an event handler that is called when the parser encounters the end of a DTD.

See also: OnDTDStart

3

OnDTDEntityExternal — event

```
property OnDTDEntityExternal : TXpDTDEntityExternalEvent
```

```
TXpDTDEntityExternalEvent = procedure(oOwner : TObject;
  bPeFlag : Boolean; sName, sPublicId, sSystemId,
  sNotationName : DOMString) of object;
```

Defines an event handler that is called when a DOCTYPE declaration contains an external entity declaration.

An external entity is data stored outside of the current document. oOwner is the TXpParser object raising the event. sName is the name of the external entity. bPEFlag is True if this is a parameter entity and False if it's a general entity. Parameter entities are entities that can be referenced inside a declaration. General entities are entities that can be used in the document reference. sPublicId is the value of the public identifier. sSystemId is the value of the system identifier. sNotationName is the value of the notation used.

See also: OnDTDEntityInternal, OnResolveEntity

OnDTDEntityInternal — event

```
property OnDTDEntityInternal : TXpDTDEntityInternalEvent
```

```
TDTDEntityInternalEvent = procedure(oOwner : TObject;
  bPeFlag : Boolean; sName, sValue : DOMString) of object;
```

Defines an event handler that is called when an internal entity declaration is found in a DTD.

An internal entity is replacement text found within the current document. oOwner is the TXpParser object raising the event. bPEFlag is True if this is a parameter entity and False if it's a general entity. Parameter entities are entities that can be referenced inside a declaration. General entities are entities that can be used in the document reference. sName is the name of the entity. sValue is the value of the entity.

See also: OnDTDEntityExternal

OnDTDExternalEnd — event

```
property OnDTDExternalEnd : TNotifyEvent
```

Defines an event handler that is called after an external DTD is parsed.

See also: OnDTDExternalStart

OnDTDExternalStart — event

```
property OnDTDExternalStart : TXpDTDExternalStartEvent
```

```
TXpDTDExternalStartEvent = procedure(oOwner : TObject;
  var sFile : DOMString) of object;
```

Defines an event handler that is called before parsing an external DTD source.

oOwner is the TXpParser object raising the event. sFile is the file name of the external DTD to parse. If the file does not specifically identify the location of the DTD, set the sFile parameter to the location of the DTD in this event handler.

See also: OnDTDExternalEnd

OnDTDNotation — event

```
property OnDTDNotation : TXpDTDNotationEvent
```

```
TXpDTDNotationEvent = procedure(oOwner : TObject;
  sNotationName, sPublicId, sSystemId : DOMString) of object;
```

Defines an event handler that is called for a NOTATION declaration inside a DTD.

A NOTATION declaration is used to declare external non-XML content, such as binary data. oOwner is the TXpParser object firing the event. sNotationName is the name of the notation. sPublicId is the value of a shared resource and sSystemID is the value of a specific resource.

External resources are identified by a public or system uniform resource identifier (URI). A public identifier is used to specify the location of a shared resource. For instance, you may want to use a public company logo (binary graphical data) in a multi-user system to ensure all users are using the same logo. Normally these types of shared resources will have a common name, such as “XYZ’sLogo.” The TXpParser user is responsible for locating public resources.

A system location is used to specify a specific file, FTP site, or Web address. A system location can be used as a backup to a public resource or as the sole source when a public resource isn't used. Valid system locations could include "c:\MyDTDs\ReportFormat.jpg", "file://c:\MyDTDs\Logo.jpg", "ftp://ftp.xyz.com/Logo.jpg", or "http://www.xyz.com/Logo.jpg."

See also: OnResolveEntity

OnDTDStart — event

```
property OnDTDStart : TXpDocTypeDeclEvent
```

```
TXpDocTypeDeclEvent = procedure(oOwner : TObject;
  sDecl, sId0, sId1 : DOMString) of object;
```

Defines an event handler that is called when the parser encounters the start of a DTD.

A DTD (document type definition) is the set of rules for validating an XML document. oOwner is the TXpParser object firing the event. sDecl is the qualified name of the document. sId0 is the document's public identifier. sId1 is the document's system identifier.

A public identifier is used to specify the location of a shared resource. For instance, you may want to use a public DTD in a multi-user system to ensure all users are using the same DTD. Normally these types of shared resources will have a common name, such as "XYZ'sReportFormat." The TXpParser user is responsible for locating public resources.

A system identifier is used to specify a specific location, such as a file, FTP site, or Web address. A system location can be used as a backup to a public resource or as the sole source when a public resource isn't used. Valid system locations could include "c:\MyDTDs\ReportFormat.dtd", "file://c:\MyDTDs\ReportFormat.dtd", "ftp://ftp.xyz.com/ReportFormat.dtd", or "http://www.xyz.com/ReportFormat.dtd."

Note that TXpParser is not a validating parser and does not validate XML documents using the DTD. TXpParser on verifies that XML documents meet the XML specification's well-formedness constraints.

See also: OnDTDEnd, OnResolveEntity

OnEndDocument — event

```
property OnEndDocument : TNotifyEvent
```

Defines an event handler that is called when the document has finished parsing.

See also: OnStartDocument

OnEndElement — event

```
property OnEndElement : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

Defines an event handler that is called when parsing of an element is complete.

oOwner is the TXpParser object raising the event. sValue is the name of the parsed element.

See also: OnStartElement

OnNonXMLEntity — event

```
property OnNonXMLEntity : TXpNonXMLEntityEvent
```

```
TXpNonXMLEntityEvent = procedure(
  oOwner : TObject; sEntityName, sPublicId, sSystemId,
  sNotationName : DOMString) of object;
```

Defines an event handler that is called when an entity is defined with NDATA.

NDATA is unparsed non-XML data, such as an image file. oOwner is the TXpParser object raising the event. sEntityName is the name of the entity being defined sPublicId is the value of the public identifier. sSystemId is the value of the system identifier.

A public identifier is used to specify the location of a shared resource. For instance, you may want to use a public company logo image in a multi-user system to ensure all users are using the same logo. Normally these types of shared resources will have a common name, such as "XYZ'sLogo." The TXpParser user is responsible for locating public resources.

A system identifier is used to specify a specific location, such as a file, FTP site, or Web address. A system location can be used as a backup to a public resource or as the sole source when a public resource isn't used. Valid system locations could include "c:\MyDTDs\ReportFormat.dtd", "file://c:\MyDTDs\Logo.jpg", "ftp://ftp.xyz.com/Logo.jpg", or "http://www.xyz.com/Logo.jpg."

sNotationName is the value of the notation used.

See also: OnResolveEntity

OnPreserveSpace — event

```
property OnPreserveSpace : TXpPreserveSpaceEvent
```

```
TXpPreserveSpaceEvent = procedure(oOwner : TObject;
  sElementName : DOMString; var bPreserve : Boolean) of object;
```

3

Default: False

Defines an event handler that is called to determine whether to preserve white space.

This event is raised for each element to determine whether the white space under the specified element is to be preserved or not. This event allows the NormalizeData property and xml:space attribute to be overridden. oOwner is the TXpParser object raising the event. Setting bPreserve to True preserves the white space. sElementName is the name of the current element.

See also: NormalizeData

OnProcessingInstruction — event

```
property OnProcessingInstruction : TXpProcessInstrEvent
```

```
TXpProcessInstrEvent = procedure(oOwner : TObject;
  sName, sValue : DOMString) of object;
```

Defines an event handler that is called for each processing instruction.

Processing instructions are instructions intended to be executed the using application, not the parser. oOwner is the TXpParser object raising the event. sName is the name of the target application. sValue is the value of data, if specified.

Note that processing instructions may appear outside the root element. This means this event may be raised anywhere between the OnStartDocument event and the OnEndDocument event.

OnResolveEntity event

```
property OnResolveEntity : TXpResolveEvent
```

```
TXpResolveEvent = procedure(oOwner : TObject;
  sName, sPublicId, sSystemId : DOMString;
  var sValue : DOMString) of object;
```

Defines an event that is called for each external text entity that occurs in the data.

This event allows external references to be resolved.

External resources are identified by a public and system uniform resource identifier (URI). A public identifier is used to specify the location of a shared resource. For instance, you may want to use a public company logo image in a multi-user system to ensure all users are using the same logo. Normally these types of shared resources will have a common name, such as "XYZ'sLogo." The TXpParser user is responsible for locating public resources.

A system identifier is used to specify a specific location, such as a file, FTP site, or Web address. A system location can be used as a backup to a public resource or as the sole source when a public resource isn't used. Valid system locations could include "c:\MyDTDs\ReportFormat.dtd", "file://c:\MyDTDs\Logo.jpg", "ftp://ftp.xyz.com/Logo.jpg", or "http://www.xyz.com/Logo.jpg."

This event allows the application to locate a public resource and retrieve its (URI) or decide between using a public resource and a system resource. After these tasks have been accomplished, the URI of the selected resource should be assigned to sValue.

oOwner is the TXpParser object firing the event. sName is the name of the entity to resolve. sPublicID is the public identifier and sSystemID is the system identifier.

See also: OnDocTypeDecl, OnDTDEntityExternal, OnDTDNotation, OnDTDStart, OnNonXMLEntity

OnStartDocument event

```
property OnStartDocument : TNotifyEvent
```

Defines an event handler that is called at the start of the document parsing.

A document, as used here, is a collection of XML structures and data. This event is called for the source document and any external dependencies.

See also: OnAttribute, OnEndDocument

OnStartElement — event

```
property OnStartElement : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

3

Defines an event handler that is called when an element start tag is encountered.

A start tag is used to identify the beginning of an element. oOwner is the TXpParser object raising the event. sValue is the name of the current element.

See also: OnEndElement

ParseDataSource — method

```
function ParseDataSource(sSource : String) : Boolean;
```

Loads and parses the specified data source.

The data source, aSource, can be a file name, HTTP URL, or an FTP site containing the XML document to parse. The function will return True on success and False if an error occurs. See ParseMemory if the XML data is in memory.

The following code example illustrates the different ways in which this method may be invoked:

```
Parser.ParseDataSource('n:\xml\tugs\manifest.xml');
Parser.ParseDataSource('file://d:\breeding\lineage.xml');
Parser.ParseDataSource('http://www.myFreeBeer.com/supports.xml');
Parser.ParseDataSource('ftp://ftp.weightloss.com/badFoodIdeas.xml');
```

Note: To use an HTTP URL or FTP location as the data source, WININET.DLL is required to be installed on Microsoft Windows-based systems. See “System Requirements” on page 5 for more information.

The following example shows how to load and parse an XML document:

```
Parser.NormalizeData := True;
if Parser.ParseDataSource(TheXMLDoc) then
  memErrors.Lines.Add('No errors')
else
  memErrors.Lines := Parser.Errors;
```

See also: ParseMemory

ParseMemory — method

```
function ParseMemory(var Buffer; aSize Longint) : Boolean;
```

Parses the specified memory buffer.

Buffer is the memory holding the XML document to parse. aSize is the number of bytes in the buffer. This function returns True if the parse is successful and False if it is not. See ParseDataSource if the XML data is located in a file, HTTP URL, or an FTP location.

The following code example parses a string containing an XML document:

```
aResult := aXMLParser.ParseMemory(aString[1], length(aString));
```

See also: ParseDataSource

Password — property

```
property Password : String
```

Default: xmlpartner@turbopower.com

Specifies the password used when retrieving an external XML document or entity from an FTP site.

See also: UserName

RaiseErrors — property

```
property RaiseErrors : Boolean
```

Default: False

Determines if the parser is to catch or raise TXpParserError exceptions.

Setting this property to False causes the parser to handle TXpParserError exceptions. Setting it to True causes exceptions to be raised to the application, where the application must respond the errors.

See also: ErrorCount, Errors, GetErrorMsg

UserName — property

```
property UserName : String
```

Default: anonymous

Specifies the user name used when retrieving an external XML document or entity from an FTP site.

See also: Password

Chapter 4: Document Object Model Core Interfaces

XMLPartner implements the core and extended interfaces in the Document Object Model (DOM) Level 1 Specification. This chapter describes the core interfaces implemented by the XMLPartner Document Object Model. For information about the extended interfaces, see Chapter 5.

Some, but not all, of the DOM Core Level 2 Specification has been implemented. XMLPartner does not implement any HTML interfaces. Table 4.1 maps the DOM core interfaces to their corresponding XMLPartner classes.

Table 4.1: *DOM core interfaces with corresponding XMLPartner classes*

DOM Interface	XMLPartner Class
DOMException	TXpDOMException
DOMImplementation	TXpDOMImplementation
DocumentFragment	TXpDocumentFragment
Document	TXpDocument
Node	TXpNode
NodeList	TXpNodeList
NamedNodeMap	TXpNamedNodeMap
CharacterData	TXpCharacterData
Attr	TXpAttribute
Element	TXpElement
Text	TXpText
Comment	TXpComment

While the event-driven model detailed in Chapter 3 is useful for a one-time scan through an XML document, the DOM builds an in-memory tree structure representing the XML document. DOM is useful when an application must do one or more of the following:

1. Use XPath expressions to find elements within a document.
2. Edit the contents of a document and save the modified document.
3. Build an XML document from scratch.

XMLPartner introduces a new class, TXpObjModel, which represents the DOM view of an XML document. Class TXpObjModel provides interfaces for loading XML documents, saving a DOM to file, tracking the use of elements and attributes, and detecting errors encountered during the parsing of an XML document.

At its heart, the XMLPartner DOM implementation uses the TXpParser class defined in Chapter 3. The TXpObjModel serves as a wrapper for the parser, tracking errors and creating various objects as the parser scans the XML document. The DOM implementation handles all the character sets recognized by the TXpParser class.

One area in which the DOM implementation differs from the event-driven parser is in the area of memory usage. As detailed in Chapter 3, the TXpParser creates a fixed size buffer for each XML document being parsed. The parser keeps only a portion of the XML document in memory at any one time. The DOM implementation keeps the entire document, and any external entities, in memory at any one time. An application designed to handle very large XML documents must keep this fact in mind.

An instance of TXpObjModel may be instantiated dynamically in code or by placing a TXpObjModel component on a form or data module.

Loading and saving documents

An instance of TXpObjModel is not the XML document. However, an instance of TXpObjModel can load an XML document and provide access to the XML document via its Document property.

The object model may load a document from a file, HTTP location, FTP location, or memory. This section provides an example of each situation. In the Windows version of XMLPartner, the loading from HTTP or FTP requires the use of WININET.DLL supplied with Microsoft Internet Explorer.

If you want the parser to strip out extra white space from your XML document, set the NormalizeData property of class TXpObjModel to True prior to loading a document from a file or memory. NormalizeData is set to False by default, which means that white space and carriage returns within elements will be stored in the text node under the element.

The following example illustrates the loading of a document from a local hard drive:

```
aObjModel.NormalizeData := True;
aObjModel.LoadDataSource('c:\docs\blackadder.xml');
```

Optionally, the file path may be prefixed with “file://” as shown in the following example:

```
aObjModel.LoadDataSource('file://c:\docs\blackadder.xml');
```

The following example illustrates the loading of a document from an HTTP source:

```
aObjModel.LoadDataSource(
  'http://www.turbopower.com/products/summary.xml');
```

The following example illustrates the loading of a document from an FTP source. Note that a user name and password must be supplied in order to log into the FTP site:

```
aObjModel.UserName := 'hal';
aObjModel.Password := 'monolith';
aObjModel.LoadDataSource(
  'ftp://ftp.coolhistory.com/pub/agincourt.xml');
```


An easy way to start a new XML document from scratch is to load the document from memory. The memory buffer usually takes the form of a widestring. The parser evaluates the buffer as though it were being read from a file. The following code example illustrates the loading of an empty document from memory:

```
function CreateEmptyDoc : TXpObjModel;
var
  aBuffer : string;
begin
  aBuffer := '<doc/>';
  Result := TXpObjModel.Create(nil);
  Result.LoadMemory(aBuffer[1], size(aBuffer));
end;
```

To save an XML document to disk, use the SaveToFile method of class TXpObjModel. The document may be written to a hard drive or an FTP site. The DOM is saved exactly as it is represented with the exception that adjacent text nodes are merged into one text node. To have the DOM make the XML more readable by indenting elements and inserting line feeds, set the FormattedOutput property to True.

The following example shows how to save the file to local disk:

```
aObjModel.FormattedOutput := True;
aObjModel.SaveToFile('c:\docs\llamaPosturing.xml');
```

The following example shows how to save the file to an FTP site:

```
aObjModel.UserName := 'buck';
aObjModel.Password := 'shoottokill';
aObjModel.SaveToFile
  ('ftp://ftp.fbi10watch.com/10MostWanted.xml');
```

Note that a user name and password must be specified in order to log into the FTP site.

Navigating the document

An object model is structured like a tree using instances of classes that inherit from TXpNode. Each XML document has a root element. To access the root element, use the DocumentElement property of the TXpObjModel or TXpDocument classes. The following example shows how to use the DocumentElement property of the TXpDocument class:

```
aRootElement := aObjMode.Document.DocumentElement;
```


The relationships between the nodes may be described using terms such as parent, child, and sibling. The TXpNode class declares properties that provide access to other nodes based upon their relationship to the current node. Table 4.2 summarizes the relational properties.

Table 4.2: *Node relational properties*

Property	Meaning
ChildNodes	If the node is of a type that would have children, this property returns the node list containing the child nodes.
FirstChild	If the node has children, returns the first child node.
LastChild	If the node has children, returns the last child node.
NextSibling	Returns the node immediately following the current node in the document tree.
ParentNode	Returns the node above the current node in the document tree.
PreviousSibling	Returns the node immediately preceding the current node in the document tree.

Given a current node, Figure 4.1 provides you with a visual picture of what nodes would be returned for each of the previously mentioned properties.

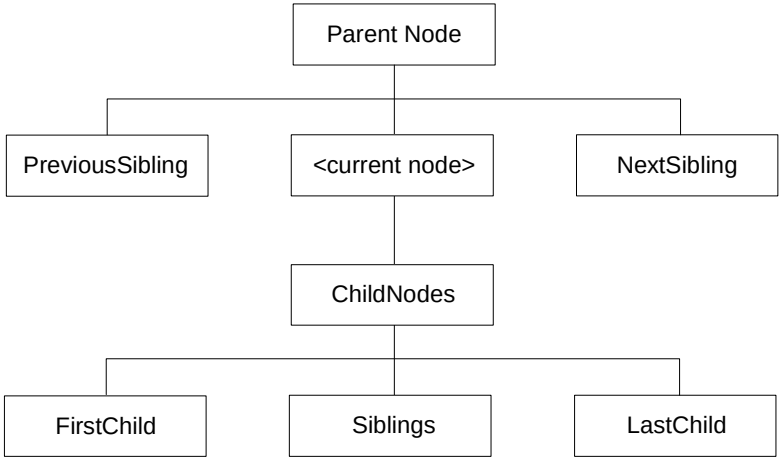


Figure 4.1: Nodes returned for relational properties

Creating documents dynamically

There are no nodes in the document tree after the DOM is first created or after a call to the ClearDocument method of TXpObjModel. Before anything of value may be done with the document tree, it must have a root element. The root element is added as a child to the document node. Following is a simple XML document:

```
<?xml version="1.0"?>
<doc/>
```

The following example creates this document in the DOM:

```
function CreateSimpleDocument : TXpObjModel
var
  aElem: TXpElement;
  aPI: TXpProcessingInstruction;
begin
  Result := TXpObjModel.Create(nil);
  aPI := Result.Document.CreateProcessingInstruction(
    'xml','version="1.0"');
  Result.Document.AppendChild(aPI);
  aPI.Release;

  aElem := Result.Document.CreateElement('doc');
  Result.Document.AppendChild(aElem);
  aElem.Release;
end;
```

Notice that instead of explicitly creating the processing instruction and element, the source code uses the CreateProcessingInstruction and CreateElement methods of class TXpDocument. Nodes, such as processing instructions and elements, are reference counted. After each node is added to the tree, the source code decrements the reference count by calling the Release method.

Adding an element to an existing element

According to the XML specification, elements may be nested within other elements. This section shows you how to nest elements using the DOM. To add a new element, you must first create the element as shown in the following example:

```
anElem := anObjModel.Document.CreateElement('address');
```

Next, add the new element to the DOM as a child of another element:

```
aParentElem.AppendChild(anElem);
```

Finally, release the element node since it is now added to the tree:

```
anElem.Release;
```

These 3 steps can be consolidated into 1 line of code by using the CreateChildElement method of the TXpElement class:

```
anElem := aParentElem.CreateChildElement('address');
```

Setting and getting attributes

Elements within the DOM may have one or more attributes. Each attribute has a name and a value. Depending upon the Document Type Definition (DTD), an attribute may have a default value. The following line of code sets the “name” attribute of an element to the value “Joe Smith”:

```
anElem.SetAttribute('name', 'Joe Smith');
```

Assuming that anElem points to an element called “contact”, this generates code that looks like this:

```
<contact name="Joe Smith"/>
```

The following line of code retrieves the value of an element’s “name” attribute:

```
sName := anElem.GetAttribute('name');
```

Preserving space

When creating documents, it is often convenient to use white space (i.e., spaces, tabs, and line feeds) to improve the readability of the documents. This white space is typically not intended for inclusion in the delivered document or object model representation. However, there are some situations in which white space should be preserved in the delivered version, for example in poetry and source code.

The handling of white space is up to the processor. However, there are 2 ways to mark up XML documents to guarantee that white space is preserved. The first is by including literal text in a CDATA section as shown in the following example:

```
<![CDATA[
  void HelloWorld()
  {
    printf("Hello world\n");
  }
]]>
```

CDATA sections are described in “TXpCDATASection Class” on page 178 and may occur anywhere that character data may occur. CDATA sections begin with “<![CDATA[” and end with “]]>”. Within a CDATA section, only the “]]>” is recognized as markup. This means that left angle brackets (<) and ampersands may occur in their literal form. In other words, the XML parser does not evaluate any of the characters between the start and end tags of the CDATA section. All white space within a CDATA section is preserved and is not stripped out by the XML parser.

The second way to guarantee that white space will be preserved is to add a special attribute named “xml:space” to an element. This attribute signals the XML parser to preserve the white space in that element. The “xml:space” attribute may have one of two values: “default”

and “preserve.” If “default” is specified, the application carries out its default white space processing. If “preserve” is specified then white space will be preserved regardless of application settings. The setting applies to all nested tags unless it the “xml:space” is specified differently at a lower level.

The following example illustrates the use of the “xml:space” attribute:

```
<code xml:space="preserve">
  void HelloWorld()
  {
    printf("Hello world\n");
  }
</code>
```

The default setting for the TXpObjModel component is to preserve all white space when parsing a document. This means that white space is included in text nodes of the document object model (DOM).

In the following example, the ‘p’ element will have 3 immediate children when the fragment is parsed into the DOM. The first child will be a text node containing “#0D#0A The Rocky Mountains are in ”. The second child will be a “state” element and the third child will hold a text node containing “#0D#0A”.

```
<p>
  The Rocky Mountains are in <state>Colorado</state>
</p>
```

To strip out non-essential white space when parsing a document, set the NormalizeData property of the TXpObjModel component to True. Using the previous example, if NormalizeData is set to the value True then the ‘p’ element will have only 2 children. The first child will hold a text node containing “The Rocky Mountains are in” and the second child will be the “state” element.

If a document isn’t parsing as expected, check the value of the NormalizeData property.

Querying the DOM

XMLPartner provides two ways for an application to search for nodes within the DOM. The result of a search is generally a list of nodes that satisfy the search conditions.

The GetElementsByTagName method of classes TXpDocument and TXpElement search for elements matching a particular tag name. The GetElementsByTagNameNS performs the same kind of search but includes a namespace URI as part of the comparison. The following example returns a list of all “item” nodes in the DOM:

```
aNodeList := aObjModel.Document.GetElementsByTagName('item');
```

The SelectNodes method of class TXpElement returns a node list containing nodes that match an XPath expression. See the XPath Version 1.0 Recommendation for details on XPath expressions. The following example returns a list of all “item” nodes in the DOM:

```
aNodeList := aObjModel.DocumentElement.SelectNodes('//item');
```

The SelectBoolean, SelectInteger, SelectNumber and SelectString methods of class TXpElement return the results of the XPath expression as the specified type.

Validating the DOM

The XMLPartner parser is not a validating parser. However, the TXpObjModel class provides a ValidateDocument method that compares the document content against the declarations in the Document Type Definition (DTD). The following example shows how to initiate the validation:

```
if aObjModel.ValidateDocument then
  ShowMessage('Document is valid')
else
  ShowMessage('Document is NOT valid');
```

The document may be invalid for a number of reasons, as documented in the OnInvalidDocument event of TXpDocument. To catch validation errors, provide a handler for the OnInvalidDocument event. The event is raised for each validation error encountered. The event specifies a code for the reason the document is invalid, the node to which the error code applies, and an output parameter to stop the validation process.

The following example shows how to define a handler for this event:

```
procedure TValidateForm.XpObjModelOnInvalidDocument(
  oOwner: TObject; wCode: Integer; oNode: TXpNode; var bStop:
Boolean);
begin
  aErrListBox.Items.Add(MapCodeToMsg(wCode) +
    ': ' + oNode.NodeName);
  { Stop only if the document is missing a required attribute. }
  bStop := (wCode = V_ATTRIBUTEREQUIRED);
end;
```

Error handling

By default, the TXpObjModel class captures errors that occur while the document is being parsed. After the document has been parsed (i.e., after the LoadDataSource or LoadMemory method has been called), the ErrorCount property returns the number of errors encountered by the parser. The Errors property provides a list of the errors.

To have errors raised to the application using an instance of TXpObjModel, set the RaiseErrors property of TXpObjModel to True.

Instances of TXpDOMException may be raised if the application performs an illegal operation while manipulating the DOM. The Code property of TXpDOMException identifies the reason for the error.

TXpObjModel Component

The TXpObjModel component represents XML as an object model in memory. The interfaces provided by the object model generally conform to the W3C Document Object Model (DOM) Level 1 Specification.

To build an object model for an XML document, use the LoadDataSource or LoadMemory methods of this component. Use LoadDataSource if the XML document resides in a file on the local file system or is available via HTTP or FTP. Use LoadMemory if the XML document resides in an in-memory buffer.

When this component loads the document, it creates an instance of class TXpDocument to represent the XML document. Use the Document property to access the instance of TXpDocument. Use the SaveToFile method to write a modified object model to disk.

Hierarchy

TComponent (VCL/CLX)

TXpObjModel (XPDOM)

Properties

BufferSize
Document
DocumentElement
ErrorCount
Errors
FormattedOutput
IdAttribute
IgnoreCase
NormalizeData
Password
RaiseErrors
TrackAttributes
TrackedAttributes
TrackedElements
TrackElements
UserName
XmlDocument

Methods

ClearDocument
GetErrorMsg
LoadDataSource
LoadMemory
SaveToFile
ValidateDocument

Events

OnFunction
OnFunctionAvailable
OnInvalidDocument
OnPreserveSpace
OnResolveEntity
OnSystemProperty
OnVariableLookup

BufferSize — property

```
property BufferSize : Integer
```

Default: 8192

Defines the number of characters the parser's buffered stream will read at any one time.

So that an application may process very large XML documents without consuming too much memory, the XML parser reads only a portion of an XML document at any one time. The BufferSize property specifies how many characters the parser may hold in memory at any one time. The larger the buffer size, the faster the performance of the parser as it requires fewer disk reads to obtain the document.

The following example illustrates the use of this property:

```
aObjModel.BufferSize := 64 * 1024;
if aObjModel.LoadDataSource('ftp://ftp.Mong.com/docs.xml') then
  ShowMessage('Document parsed successfully.');
```

See also: LoadDataSource

ClearDocument — method

```
procedure ClearDocument;
```

Destroys the current XML object model.

Use this method to free the object model contained by TXpObjModel. The LoadDataSource and LoadMemory methods automatically call this method, so it is not necessary to call ClearDocument prior to loading a new XML document.

See also: Document, LoadDataSource, LoadMemory

Document — read-only property

```
property Document : TXpDocument
```

Returns the document object of the XML object model.

Use this property to obtain the document object of the XML object model. The document object provides access to the contents of the XML document.

See also: ClearDocument, TXpDocument

DocumentElement — read-only property

```
property DocumentElement : TXpElement
```

Provides direct access to the document's root element.

This property is for the convenience of the developer, providing access to the instance of TXpElement that is the root element of the document.

ErrorCount — read-only property


```
property ErrorCount : Integer
```

Returns the number of errors found while parsing an XML document.

Use this property to determine if any errors occurred while the XML document was parsed. If no errors occurred, this property returns the value zero. To access the error messages, use the Errors property or the GetErrorMsg method. Note that both Errors and GetErrorMsg are zero-based therefore their upper bound is equivalent to ErrorCount minus one.

If the RaiseErrors property is set to True, ErrorCount will be non-zero and Errors will contain the error message related to the exception.

The following code example illustrates the use of this property:

```
{ Any errors during parsing? }
if aXMLObj.ErrorCount = 0 then
  lblStatus.Caption := 'Document is well-formed.'
else
  lblStatus.Caption := 'Encountered errors while parsing.';
```

See also: Errors, GetErrorMsg, RaiseErrors

Errors — read-only property

```
property Errors : TStringList
```

Contains the errors reported while parsing an XML document.

Use this property to access the errors encountered while parsing an XML document. The ErrorCount property indicates whether any errors were encountered and specifies the upper bound for the Errors list. The Errors list is base zero so the upper bound of the list is ErrorCount minus one.

If RaiseErrors is set to True then this property will contain the error message associated with the exception. Errors may also be retrieved via the GetErrorMsg method.

The following code example illustrates the use of this property:

```
{ Any errors during parsing? }
if aXMLObj.ErrorCount > 0 then
  { Yes.  Add them to the list box. }
  for anIndex := 0 to pred(aXMLObj.ErrorCount) do
    lbErrors.Items.Add(aXMLObj.Errors[anIndex]);
```

See also: ErrorCount, GetErrorMsg, RaiseErrors

FormattedOutput — property

```
property FormattedOutput : Boolean
```

Default: False

Indicates whether a saved XML document is to include line breaks and indents to improve readability.

Use this property to improve readability of a saved XML document. By default, XML documents are written as is via the SaveToFile method. They do not include any additional line breaks or spacing. Setting this property to True inserts a carriage return and line feed after each element, comment, and processing instruction node. In addition, each nested element, processing instruction, and comment nodes is indented two spaces per level of the node.

To illustrate this property, the following XML document is read into the XML editor provided in the Examples subfolder:

```
<?XML VERSION="1.0" ENCODING="UTF-8" ?>
<!DOCTYPE TIRE []>
<TIRE MODEL="X-One" SIZE="P185/70R15">
<MANUFACTURER NAME="Michelin"
HREF="http://www.michelin.com"/><DESCRIPTION>
The all-season T speed-rated touring tire with Radial XSE
Technology to deliver the ultimate in mileage and wet traction
for today's performance touring and domestic luxury automobiles.
<!-- Available end of June --></DESCRIPTION><UTQC TREADWEAR="620"
TRACTION="A" TEMPERATURE="B" Durability="AA"/><PRICE
UNIT="Dollars">265.78 <!-- not including tax --> </PRICE></TIRE>
```

When the XML document is saved with FormattedOutput set to True, the XML document now appears as follows:

```
<?XML VERSION="1.0" ENCODING="UTF-8" ?>
<!DOCTYPE TIRE [
]>
<TIRE MODEL="X-One" SIZE="P185/70R15">
  <MANUFACTURER NAME="Michelin" HREF="http://www.michelin.com"/>
  <DESCRIPTION>The all-season T speed-rated touring tire with
Radial XSE Technology to deliver the ultimate in mileage and wet
traction for today's performance touring and domestic luxury
automobiles.<!-- Available end of June --></DESCRIPTION>
  <UTQC TREADWEAR="620" TRACTION="A" TEMPERATURE="B"
Durability="AA"/>
  <PRICE UNIT="Dollars">265.78<!-- not including tax --></PRICE>
</TIRE>
```

See also: SaveToFile

GetErrorMsg — method

```
function GetErrorMsg(wIdx : Integer) : DOMString;
```

Returns the specified error message.

Use this method to retrieve an error message encountered during parsing. First, use the ErrorCount property to determine if any errors occurred. If ErrorCount is greater than zero, this method and the Errors property provide access to the error messages.

wIdx is base zero and is the index of the error message to retrieve. If RaiseErrors is set to True, the errors list will contain the error message associated with the exception.

The following code example illustrates the use of this property:

```
{ Any errors during parsing? }
if aXMLObj.ErrorCount > 0 then
  { Yes.  Add them to the list box. }
  for anIndex := 0 to pred(aXMLObj.ErrorCount) do
    lbErrors.Items.Add(aXMLObj.GetErrorMsg(anIndex));
```

See also: ErrorCount, Errors, RaiseErrors

IdAttribute — property

```
property IdAttribute : DOMString
```

Default: “id”

Specifies the name of the XML document’s ID attribute.

Use this property to identify which attribute in the XML document is to be used as its ID attribute.

See also: Document

IgnoreCase — property

```
property IgnoreCase : Boolean
```

Default: False

Controls the case-sensitivity of searches within the Document Object Model (DOM).

By default, all searches within the DOM are case-sensitive. Set this property to True to perform case-insensitive searches in methods such as the GetElementsByTagName method of the TXpDocument class and the SelectNodes method of the TXpElement class.

See also: TXpDocument.GetElementsByTagName, TXpElement.SelectNodes

LoadDataSource — method

```
function LoadDataSource(sSource : String) : Boolean;
```

Retrieves and parses an XML document from local disk, HTTP, or FTP.

Use this method to load and parse an XML document. The XML document may reside on a local hard drive or it may be accessible via HTTP or FTP. To parse an XML document that is already in memory, use the LoadMemory method.

sSource is the name of the XML document and must be properly formatted based upon the location of the document. If the XML document is accessed via HTTP or FTP, the entire document is copied to the computer’s temporary directory before it is parsed. The document is deleted once this method completes.

If the document is loaded and successfully parsed, this method returns True. Otherwise, it returns False. If this method returns False, use the ErrorCount and Errors property to access the error information.

The following code example illustrates the different ways in which this method may be invoked:

```
aObjModel.LoadDataSource('n:\xml\tugs\manifest.xml');
aObjModel.LoadDataSource('file://d:\breeding\lineage.xml');
aObjModel.LoadDataSource('http://www.myPlace.com/supports.xml');
aObjModel.LoadDataSource('ftp://ftp.food.com/badFoodIdeas.xml');
```

This method does not load the entire document into memory. Instead, it uses a buffered stream to store only a portion of the document at any one time. To control the size of the buffer, use the BufferSize property. If the document references external entities, the external entities are also read using a buffered stream, one per external entity. This scheme allows an application to handle very large XML documents without using a large amount of memory.

Once this method has parsed the document, the document's object model may be accessed via the Document property.

See also: BufferSize, Document, ErrorCount, Errors, LoadMemory

LoadMemory — method

```
function LoadMemory(var Buffer; aSize : Longint) : Boolean;
```

Parses an in-memory XML document.

Use this function to parse an XML document that has already been loaded into memory. Buffer is a variable containing the XML document. For example, Buffer could be a string, a widestring, or a dereferenced pointer. The parser scans Buffer for a prolog element identifying the character set and parses Buffer appropriately. Parameter assize is the length of the data in the buffer.

This function returns True if the document was successfully parsed. Otherwise, it returns False. If this function returns False, use the ErrorCount and Errors property to access the error information.

Once this method has parsed Buffer, the document's object model may be accessed via the Document property.

The following code example illustrates the use of this function:

```
aBuffer := '<Description>My XML document</Description>';
if aXMLObjModel.LoadMemory(aBuffer[1], length(aBuffer)) then
  ShowMessage('Document parsed successfully.');
```

See also: Document, ErrorCount, Errors, LoadDataSource

```
property NormalizeData : Boolean
```

Default: True

Determines whether extra white space is omitted from the parsed document.

By default, extra white space is removed from character data as the document is parsed. Extra white space is comprised of any leading or trailing white space characters and multiple white space characters found within the character data. To leave character data as is, set this property to False. When this property is set to False, stripping of white space may be controlled using the OnPreserveSpace event.

The following example shows the results of normalization:

Before:

```
"    The dog    jumped over    the cat.     "
```

After:

```
"The dog jumped over the cat."
```

Normalized data can cause problems if leading and trailing spaces are used to format character data from multiple elements. See the following XML document for an example:

```
<Contents>
  <P>
    <Font FontName="Arial" PointSize="10" Color="#000000">
      The importance of properly formatting data </Font>
    <Font FontName="Arial" PointSize="10" Color="#000000">
      <B>can not </B>
    </Font>
    <Font FontName="Arial" PointSize="10" Color="#000000">
     be over emphasized.
    </Font>
  </P>
</Contents>
```

Using the previous XML document, the result of combining character data from the three elements produces:

```
The importance of properly formatting datacan notbe over
emphasized.
```

Note that there isn't a space between "data" and "can" or "not" and "be" although there is spacing to account for combining the strings. The solution is to use the xml:space attribute as follows:

```
<Contents>
  <P>
    <Font FontName="Arial" PointSize="10" Color="#000000"
     xml:space = 'preserve'>The importance of properly formatting
data </Font>
    <Font FontName="Arial" PointSize="10" Color="#000000"
     xml:space = 'preserve'>
      <B>can not </B></Font>
    <Font FontName="Arial" PointSize="10" Color="#000000"
     xml:space = 'default'>be over emphasized.</Font>
  </P>
</Contents>
```

See also: OnPreserveSpace

OnFunction — event

```
property OnFunction : TXpFunctionEvent
```

```
TXpFunctionEvent = procedure(oOwner : TObject;
  sFunctionName : DOMString; oArguments : TStringList;
  var sResult : DOMString) of object;
```

Defines an event handler called when a user function is referenced in an XPATH string.

Use this event to implement a user function present in the XML document. oOwner is the instance of TXpElement containing the XPATH. sFunctionName is the name of the user function. oArguments is an instance of TStringList containing the arguments for the function.

If the handler recognizes the function, the handler should evaluate the function with the given arguments and place the result into variable sResult.

See also: OnFunctionAvailable

OnFunctionAvailable event

```
property OnFunctionAvailable : TXpFunctionAvailableEvent
```

```
TXpFunctionAvailableEvent = procedure(oOwner : TObject;
  sFunctionName : DOMString; var bAvailable : Boolean);
```

Defines an event handler that is called when the standard XPath function named “function-available” is encountered.

4

The XPath function named function-available determines whether or not an extension function is available. Use this method to indicate whether certain user functions are available. oOwner is the instance of TXpElement associated with the function-available call. sFunctionName is the name of the user function of interest. If the user function is supported, set bAvailable to True. Otherwise, set bAvailable to False.

See also: OnFunction

OnInvalidDocument event

```
property OnInvalidDocument: TXpInvalidDocumentEvent
```

```
TXpInvalidDocumentEvent = procedure(oOwner : TObject;
  wCode : Integer; oNode : TXpNode; var bStop : Boolean) of object;
```

Declares an event handler that is called during the validation of a document with the purpose of reporting reasons a document is invalid.

When using the ValidateDocument method, the DOM raises this event when it encounters an error in the document. oOwner is the instance of TXpObjModel performing the validation. wCode is a code identifying the problem in the document. oNode is the node to which wCode applies.

To stop validation of the document, set bStop to True. Otherwise, leave it set to False and validation continues after the event.

wCode may have the following values:

Constant	Value	Meaning
V_NODTD	1	No DTD for the document
V_BADDTD	2	Bad DTD structure
V_NODOCUMENT	3	No document to check
V_MISMATCH	4	Document name does not match DTD name
V_ELEMENTNOTDEFINED	5	Element name not defined in DTD
V_ELEMENTNOTEMPTY	6	Element declared as empty but not empty
V_ELEMENTNOTCONTENT	7	Element not allowed as content
V_ELEMENTCHILDMISSING	8	Required child element missing from element
V_ELEMENTCHILDNOTEXPECTED	9	Encountered child that was not expected
V_ELEMENTATTRNOTDEFINED	10	Attributes for element are not defined
V_ATTRIBUTEFIXED	11	Cannot change a fixed attribute
V_ATTRIBUTENOTENUM	12	Attribute not in enumeration
V_ATTRIBUTEREQUIRED	13	Attribute is required

The following example shows how to define a handler for this event:

```
procedure TValidateForm.XpObjModelOnInvalidDocument
  (oOwner : TObject; wCode : Integer; oNode : TXpNode;
  var bStop : Boolean);
begin
  aErrListBox.Items.Add(MapCodeToMsg(wCode) +
    ' : ' + oNode.NodeName);
  { Stop only if the document is missing a required attribute. }
  bStop := (wCode = V_ATTRIBUTEREQUIRED);
end;
```

See also: ValidateDocument

OnPreserveSpace — event

```
property OnPreserveSpace : TXpPreserveSpaceEvent
```

```
TXpPreserveSpaceEvent = procedure(oOwner : TObject;
  sElementName : DOMString; var bPreserve : Boolean) of object;
```

Defines an event handler that is called to determine if white space should be preserved.

This event is called for each element in the XML document as it is parsed into the object model. Use this event to indicate whether or not white space should be preserved in the element. This event is also useful for overriding the behavior specified via the NormalizeData property.

oOwner is the instance of TXpParser that is parsing the document. sElementName is name of the element being parsed. If white space is to be preserved, set bPreserve to True. Otherwise, set it to False.

See also: NormalizeData

OnResolveEntity — event

```
property OnResolveEntity : TXpResolveEvent
```

```
TXpResolveEvent = procedure(oOwner : TObject;
  sName, sPublicId, sSystemId : DOMString;
  var sValue : DOMString) of object;
```

Defines an event handler that is called to resolve the location of an external entity.

Use this event to override the public and system identifiers of an external entity. For example, if an XML document contained relative paths to external XML documents (e.g., "..\AntEater\CR335.xml") then this event could be used to convert the references to absolute paths (e.g., "e:\Support\AntEater\CR335.xml").

oOwner is the instance of TXpParser that is parsing the document. sName is the name of the entity being parsed. sPublicID is the public identifier of the entity and sSystemID is the entity's system identifier. Place the new system identifier in sValue.

OnSystemProperty — event

```
property OnSystemProperty : TXpVariableLookupEvent
```

```
TXpVariableLookupEvent = procedure(oOwner : TObject;
  sVarName : DOMString; var sVarValue : DOMString) of object
```

Declares an event handler that is called when the standard XSLT system-property function is encountered.

Override this event in order to provide values for system properties required by the XML document. oOwner is the instance of TXpObjModel raising the event. sVarName specifies the name of the system property required. Place the value of the system property in sVarValue.

OnVariableLookup — event

```
property OnVariableLookup : TXpVariableLookupEvent
```

```
TXpVariableLookupEvent = procedure(oOwner : TObject;
  sVarName : DOMString; var sVarValue : DOMString) of object
```

Defines an event handler that is called when a variable is referenced in an XPath expression.

Use this event to supply a value for a variable referenced in an XPath expression. oOwner is the instance of TXpObjModel evaluating the document. sVarName is the name of the referenced variable. Place the value of the variable in sVarValue.

Password — property

```
property Password : String
```

Default: “xmlpartner@turbopower.com”

Specifies the password to be used when retrieving an XML document from an FTP site.

When retrieving an XML document from an FTP site or saving an XML document to an FTP site, specify the password for the FTP site using this property.

See also: SaveToFile, UserName

RaiseErrors — property

```
property RaiseErrors : Boolean
```

Default: False

Indicates whether parsing errors should be raised as an exception.

To have parsing errors raised as an exception, set this property to True. Otherwise, the errors are accumulated and made available through the ErrorCount and Errors properties once the parsing has completed.

See also: ErrorCount, Errors, GetErrorMsg

SaveToFile — method

```
function SaveToFile(sFile : String) : Boolean;
```

Writes the object model to local file or FTP site.

Use this method to write a parsed XML object to a file on disk or FTP site. When writing to an FTP site, the UserName and Password properties must be specified. The written document is well formed. This function returns True if the file was successfully written. Otherwise, it returns False. Use the ErrorCount and Errors properties to determine the problem encountered.

sFile is the output filename and may have one of the following formats:

```
e:\inventory\generators.xml
file://e:\inventory\generators.xml
ftp://ftp.powerup.com/inventory/generators.xml
```

See also: ErrorCount, Errors, Password, UserName, XmlDocument

TrackAttributes — property

```
property TrackAttributes : Boolean
```

Default: False

Enables tracking of all attributes parsed in a document.

To maintain a list of all attribute names found in a document, set this property to True before parsing the document. After the document has been parsed, access the attribute names via the TrackedAttributes property.

See also: TrackedAttributes

TrackedAttributes — read-only property

```
property TrackedAttributes : TStringList
```

Provides access to the list of attribute names found while parsing the document.

If the TrackAttributes property was set to True prior to parsing a document, use this property to access the attributes. The attribute names are sorted alphabetically and the list does not contain any duplicates.

See also: TrackAttributes

TrackedElements — read-only property

```
property TrackedElements : TStringList
```

Provides access to the list of element names found while parsing the document.

If the TrackElements property was set to True prior to parsing a document, use this property to access the element names. The element names are sorted alphabetically and the list does not contain any duplicates.

See also: TrackElements

TrackElements

```
property TrackElements : Boolean
```

Default: False

Enables tracking of all elements parsed in a document.

To maintain a list of all element names found in a document, set this property to True before parsing the document. After the document has been parsed, access the element names via the TrackedElements property.

See also: TrackedElements

UserName — property

```
property UserName : String
```

Default: “anonymous”

Specifies the user name to use when retrieving an external entity from an FTP site.

When retrieving an XML document from an FTP site or saving an XML document to an FTP site, specify the user name for the FTP site using this property.

See also: SaveToFile, Password

ValidateDocument — method

```
function ValidateDocument : Boolean;
```

Validates the current DOM against a stored DTD.

Use this function to validate the object model against a Document Type Document (DTD). When this function encounters a validation error within the document, it raises the OnInvalidDocument event. This function returns True if the document is valid or False if at least one error was encountered.

The document must have a DTD. Otherwise, the OnInvalidDocument event is raised with error code V_NODTD.

See also: OnInvalidDocument

XmlDocument — property

```
property XmlDocument : DOMString
```

Returns the DOM as a well-formed XML document.

Use this property to retrieve an in-memory version of the XML document. For very large XML documents, this may use a large amount of memory since the application will be using memory to store both the DOM and the string returned by this function.

See also: SaveToFile

TXpDOMException Class

TXpDOMException is the exception class used by TXpObjModel to report exceptions as listed in the DOM specification. TXpDOMException is identical to the standard TException class except that TXpDOMException also includes a Code property. The Code property returns an error code that indicates the type of problem encountered.

Hierarchy

TException (VCL/CLX)

    TXpDOMException (XPDOM)

Properties

    Code

Code — read-only property

```
property Code : EXpExceptionCode
```

```
EXpExceptionCode = (ecNone, ecIndexSizeErr, ecWStringSizeErr,
  ecHierarchyRequestErr, ecWrongDocumentErr, ecInvalidNameErr,
  ecNoDataAllowedErr, ecNoModificationAllowedErr, ecNotFoundErr,
  ecNotSupportedErr, ecInuseAttributeErr, ecInvalidParamErr,
  ecExpectedToken);
```


Default: ecNone

Identifies the situation that caused the exception.

After trapping an exception of this type, use the Code property to determine the reason for the exception. The following table lists the meaning of the error codes:

Value	Meaning
ecNone	No exception has occurred. This is the default value for the Code property and should not be encountered during normal use.
ecIndexSizeErr	Raised if the specified offset is negative or greater than the number of characters available in the operation, or if a count is negative.
ecWStringSizeErr	Raised if the specified range of text does not fit in a DOMString variable.
ecHierarchyRequestErr	Raised if a node is inserted somewhere it doesn't belong.
ecWrongDocumentErr	Raised if a node involved in an operation does not have the same owner document as the other node. For example, this error is raised when appending a node owned by document A into the child list of an element owned by document B.
ecInvalidNameErr	Raised if an invalid character is specified, such as in a name.
ecNoDataAllowedErr	Raised if data is specified for a node which does not support data.
ecNoModificationAllowedErr	Raised if an attempt is made to modify a read-only object.

Value	Meaning
ecNotFoundErr	Raised if an attempt was made to reference a node in a context in which it does not exist. For example, using the TXpNode.InsertBefore method to insert a node before a node that is not a child of the current element will cause this error to be raised.
ecNotSupportedErr	Raised if the implementation does not support the type of object requested.
ecInuseAttributeErr	Raised if an attribute is already used by another element. Clone the attribute before it is re-used in another element.
ecInvalidParamErr	Raised if an invalid parameter is specified for an operation.

TXpDOMImplementation Class

The TXpDOMImplementation class implements the DOMImplementation interface as described in the DOM specification. It implements the required HasFeature method and also provides methods for creating a new document and new document type.

An instance of TXpDOMImplementation is automatically created for each document. Rarely, if ever, will an independent instance of TXpDOMImplementation need to be created. Instead, it is easiest to use an XML document's DOMImplementation via the DomImplementation property of the TXpDocument class.

Use the HasFeature method of this class to determine if the XML processor supports a DOM feature set. Use the CreateDocument and CreateDocumentType methods to create new XML documents and DTDs, respectively.

Hierarchy

TObject (VCL/CLX)

TXpDOMImplementation (XPDOM)

Methods

CreateDocument | CreateDocumentType | HasFeature

CreateDocument — method

```
function CreateDocument(sNamespaceURI, sQualifiedName : DOMString;
  oDocumentType : TXpDocumentType) : TXpDocument;
```

Creates an XML document object.

Use this method to create an XML document object having a given name and DTD. sNamespaceURI is the default namespace of the document. It is an optional parameter. If no default namespace is required then set sNamespaceURI to an empty string.

sQualifiedName is the qualified name of the document. This method creates a root element with the given name and attaches it to the new document.

oDocumentType is the DTD to associate with the document. If no DTD is available for the document, set oDocumentType to the value nil. Otherwise, if a DTD is supplied then it is attached to the new document.

This function returns an instance of TXpDocument.

See also: CreateDocumentType

CreateDocumentType — method

```
function CreateDocumentType(
  sQualified, sPublicID, sSystemID : DOMString) : TXpDocumentType;
```

Creates an empty Document Type Definition (DTD).

Use this method to create an empty DTD for use with the CreateDocument method. sQualified is the qualified name of the DTD and is assigned to the DTD's NodeName property. sPublicID and sSystemID are the DTD's public and system identifiers, respectively. To create an internal DTD, set these parameters to an empty string. To create an external DTD, specify a value for sSystemID. If the external DTD has a public identifier then specify a value for sPublicID.

This function returns an instance of TXpDocumentType.

See also: CreateDocument

```
function HasFeature(sFeature, sVersion : DOMString) : Boolean;
```

Indicates whether or not the DOM implementation supports a specific feature.

Use this method to determine if the DOM implementation supports a specific package. This function returns the True if sFeature has the value “XML” and sVersion has the value “1.0” or “2.0.”

TXpDocumentFragment Class

The TXpDocumentFragment class implements the DocumentFragment interface as defined in the DOM specification. A document fragment is a “lightweight” or “minimal” document. It may be used for tasks such as moving a portion of a document to another document or moving a portion of a document from one part of the document tree to another.

A document fragment may contain zero or more child nodes. The document fragment as a whole does not need to be well formed. For example, there does not need to be a root element. However, its children must follow the rules imposed upon well-formed XML parsed entities.

In one sense, the document fragment acts as a container for other nodes. If a document fragment is inserted into a tree, the child nodes of the document fragment and not the document fragment are inserted into the tree.

This class does not introduce any new properties, methods, or events.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 139
  - TXpDocumentFragment (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

TXpNode Class

TXpNode is the foundational class for the entire Document Object Model (DOM). It implements the Node interface as described in the DOM specification. An instance of TXpNode represents a single piece of information in the document tree. Almost all classes in the DOM inherit from TXpNode. Applications using XMLPartner should not find it necessary to create instances of TXpNode.

While all classes inheriting from TXpNode expose methods for dealing with child nodes, not all such classes may have child nodes. For example, the TXpText class may not have child nodes. Adding a child node to such a class results in the raising of an instance of TXpDOMException.

The properties NodeName, NodeType, NodeValue, and Attributes are declared in this class to allow the retrieval of node information without casting to a specific node interface (e.g., TXpText). Use the NodeName property to access the node's name. Use the NodeType property to determine what type of node is being accessed. Use the NodeValue property to obtain the value associated with a node such as a text node's text or an entity reference's entity name. The Attributes property provides access to the attributes associated with the node.

In cases where there is no obvious mapping of these attributes for a specific NodeType (e.g., NodeValue for a TXpElement or attributes for a TXpComment), the property returns a "null" value.

Node context

Each node may belong to only one instance of TXpDocument. If a factory method of TXpDocument is used to create a node, the new node is associated with the document even though it is not yet a part of the document's tree. Use the OwnerDocument property to access the node's owning document. To change the node's owning document, use the ForceOwnerDocument method. Some methods that operate upon instances of TXpNode may switch the owning document during the operation, or may expect the owning document to be correct, before performing the operation.

Reference counting

Instances of TXpNode are reference counted. The node's reference count is automatically incremented when the node is created and when the node is added to an object model. When a node's reference count reaches zero, the node is freed. After creating and using a node, be sure to call its Release method.

Hierarchy

TObject (VCL/CLX)

TXpNode (XPDOM)

Properties

Attributes
ChildNodes
FirstChild
LastChild
LevelCode
LocalName
NamespaceURI
NextSibling
NodeID
NodeName
NodeStringType
NodeType
NodeValue
OwnerDocument
ParentNode
Prefix
PreviousSibling
Text
XmlDocument

Methods

AddRef
AppendChild
CloneNode
ForceOwnerDocument
GetChildNodesByNodeType
GetNodesByNodeType
HasAttributes
HasChildNodes
InsertBefore
IsAfter
Release
RemoveAll
RemoveChild
ReplaceChild
Supports

Reference Section

AddRef — method

```
procedure AddRef;
```

Increases the node's reference count.

Nodes are reference counted. Use this method to increment the node's reference count. Use the Release method to decrement the node's reference count. AddRef is automatically called when the node is created. When a node is added to the document tree via AppendChild or InsertBefore, this method is called again.

When the reference count drops to zero, the node is freed. When you have finished adding a node to a tree, be sure to call the Release method.

See also: Release

AppendChild — method

```
procedure AppendChild(oNewChild : TXpNode);
```

Adds the specified node to the end of the node's list of children.

Use this method to append a node to the node's list of children. To insert a node prior to an existing child node, use the InsertBefore method.

If the node is already in the list of children, it is removed from the list and then appended to the list. oNewChild is the node to be appended. The following example illustrates the use of this method:

```
aNode := TXpText.Create;
aParentNode.AppendChild(aNode);
```

See also: InsertBefore

Attributes — read-only property

```
property Attributes: TXpNamedNodeMap
```

Provides access to the node's attributes.

Use this method to access the node's attributes. If the node has no attributes, this property returns nil. Otherwise, it returns the instance of TXpNamedNodeMap containing the attributes.

See also: TXpNamedNodeMap

ChildNodes — read-only property

```
property ChildNodes : TXpNodeList
```

Provides access to the child nodes of the current node.

Use this property to obtain a list of the child nodes directly connected to this node. If the node has no children, this property returns an instance TXpNodeList containing no nodes. Use the HasChildNodes to determine node has children.

Changes made to the nodes within TXpNodeList take immediate effect.

See also: FirstChild, HasChildNodes, LastChild, NextSibling

CloneNode — virtual, abstract method

```
function CloneNode(
  bDeep : Boolean = True) : TXpNode; virtual; abstract;
```

Defines a virtual, abstract method for the cloning of nodes.

Classes inheriting from TXpNode must implement this method. Use this method to make a copy of a node. The duplicate node does not have a parent node (i.e., it is not attached to the original node's parent).

If the value of bDeep is True, the node and its child nodes are cloned. If the value of bDeep is False, only the node and its attributes are copied.

This method returns the clone of the node.

FirstChild — read-only property

```
property FirstChild : TXpNode
```

Returns the first node in a node's list of child nodes.

Use this method to retrieve the first child node in a node's list of children. If the node has no children, this property returns nil.

The following code example illustrates the use of this method:

```
aChild := aNode.FirstChild;
while (aChild <> nil) do begin
  WriteLn(Log, aChild.NodeName);
  aChild := aChild.NextSibling;
end;
```

See also: HasChildNodes, LastChild, NextSibling, PreviousSibling

ForceOwnerDocument method

```
procedure ForceOwnerDocument(oNode : TXpNode);
```

Forces the specified node and its descendants to accept the current node as their owning document.

Each node in the object model has an owning document of type TXpDocument, accessible via the OwnerDocument property. Use this method to set the owning document of a node and its child nodes. oNode is the document whose OwnerDocument is to be changed. The current node must be of type TXpDocument.

This method is useful when copying a node and its children from one DOM to another. After the nodes have been cloned, use this method to set the OwnerDocument of the clones. The following example illustrates this technique:

```
procedure CopyNodeToDOM(
  sourceNode : TXpNode; targetDocument : TXpDocument);
begin
  aNode := sourceNode.CloneNode(True);
  aNode.ForceOwnerDocument(targetDocument);
end;
```

See also: OwnerDocument

GetChildNodesByNodeType method

```
function GetChildNodesByNodeType(wType : Integer): TXpNodeList;
```

Retrieves a list of immediate child nodes matching the specified node type.

Use this method to find all immediate child nodes that match the specified node type. This method does not search deeper than the immediate child nodes. wType specifies the node type. If none of the child nodes match the specified node type, this function returns an instance of TXpNodeList containing no nodes.

To search the child nodes in depth, use the GetNodesByNodeType method.

The legal values for wType are listed in the following table:

Constant	Value	Node Type
ELEMENT_NODE	1	The node is a TXpElement.
ATTRIBUTE_NODE	2	The node is a TXpAttribute.
TEXT_NODE	3	The node is a TXpText.
CDATA_SECTION_NODE	4	The node is a TXpCDATASection.
ENTITY_REFERENCE_NODE	5	The node is a TXpEntityReference.
ENTITY_NODE	6	The node is a TXpEntity. Note that this value is not used in the object model.
PROCESSING_INSTRUCTION_NODE	7	The node is a TXpProcessingInstruction.
COMMENT_NODE	8	The node is a TXpComment.
DOCUMENT_NODE	9	The node is a TXpDocument.
DOCUMENT_TYPE_NODE	10	The node is a TXpDocumentType.
DOCUMENT_FRAGMENT_NODE	11	The node is a TXpDocumentFragment.
NOTATION_NODE	12	The node is a TXpNotation.

See also: ChildNodes, GetNodesByNodeType, HasChildNodes

```
function GetNodesByNodeType(wType : Integer) : TXpNodeList;
```

Retrieves a list of the child nodes matching the specified node type.

Use this method to find all child nodes that match the specified node type. This method searches the child nodes in depth. wType specifies the node type. If none of the child nodes match the specified node type, this function returns an instance of TXpNodeList containing no nodes.

To search only the immediate child nodes, use the GetChildNodesByNodeType method.

The legal values for wType are listed in the following table:

Constant	Value	Node Type
ELEMENT_NODE	1	The node is a TXpElement.
ATTRIBUTE_NODE	2	The node is a TXpAttribute.
TEXT_NODE	3	The node is a TXpText.
CDATA_SECTION_NODE	4	The node is a TXpCDATASection.
ENTITY_REFERENCE_NODE	5	The node is a TXpEntityReference.
ENTITY_NODE	6	The node is a TXpEntity. Note that this value is not used in the object model.
PROCESSING_INSTRUCTION_NODE	7	The node is a TXpProcessingInstruction.
COMMENT_NODE	8	The node is a TXpComment.
DOCUMENT_NODE	9	The node is a TXpDocument.
DOCUMENT_TYPE_NODE	10	The node is a TXpDocumentType.
DOCUMENT_FRAGMENT_NODE	11	The node is a TXpDocumentFragment.
NOTATION_NODE	12	The node is a TXpNotation.

See also: ChildNodes, GetChildNodesByNodeType, HasChildNodes

HasAttributes — method

```
function HasAttributes : Boolean;
```

Returns True if the node has any attributes.

Use this method to determine if a node has any attributes. This method returns True if the node has one or more attributes. Otherwise, it returns False.

HasChildNodes — method

```
function HasChildNodes : Boolean;
```

Returns True if the node has any child nodes.

Use this method to determine if a node has any child nodes. This method returns True if the node has one or more child nodes. Otherwise, it returns False.

See also: ChildNodes

InsertBefore — method

```
procedure InsertBefore(oNewChild, oRefChild: TXpNode);
```

Inserts a node before another child node of the current node.

Use this method to insert a node before a specific child of the node. oNewChild is the node to be inserted. After being inserted, oNewChild becomes a child of the node. oRefChild is the existing child node before which the new child is to be inserted. If oNewChild is already a child of the node, it is removed from the node's list of children and then re-inserted into the specified position.

If oNewChild is an instance of TXpDocumentFragment then all of oNewChild's children are inserted, in their existing order, before oRefChild.

Note: oNewChild must belong to the same OwnerDocument as its new parent node. If the OwnerDocument is not the same then an instance of TXpDomException having error code xpecWrongDocumentErr is raised.

The following code example illustrates the use of this method:

```
aParentNode.InsertBefore(aNewNode, aChildNode);
```

See also: AppendChild

IsAfter — method

```
function IsAfter(oNode : TXpNode) : Boolean;
```

Returns True if the current node follows the specified node.

Use this method to determine if the current node occurs in the document object model after another node. oNode is the boundary node. This method returns True if the current node is after oNode. Otherwise, it returns False.

The following example illustrates the use of this method:

```
if aNode.IsAfter(aChildNode) then
  aNode.ParentNode.RemoveChild(aNode);
  aChildNode.ParentNode.InsertBefore(aNode, aChildNode);
  aNode.Release;
  aChildNode.Release;
end;
```

LastChild — read-only property

```
property LastChild : TXpNode
```

Returns the last node in a node's list of child nodes.

Use this method to retrieve the last child node in a node's list of children. If the node has no children, this property returns nil.

The following code example illustrates the use of this method:

```
aChild := aNode.LastChild;
while (aChild <> nil) do begin
  WriteLn(Log, aChild.NodeName);
  aChild := aChild.PreviousSibling;
end;
```

See also: FirstChild, HasChildNodes, NextSibling, PreviousSibling

LevelCode — read-only property

```
property LevelCode : DOMString
```

Returns an identifier that uniquely identifies the node's location within the object model.

This property returns a string code identifying the node's position within the tree. The string code contains a period-separated list of numbers where the first number is the node's position on level 1, the second number is the node's position on level 2, and so on.

For example, a node having LevelCode equal to "1.5" is the fifth child of the first node in the document. A node having LevelCode equal to "2.1.3" is the third child of the first node on the second level.

LocalName — read-only property

```
property LocalName : DOMString
```

Returns the right-hand side of a node's name.

When dealing with nodes whose name is qualified with a namespace (e.g., "cereal:SugarBombs"), use this property to retrieve the node's local name. In the previous example, the local name is "SugarBombs." If a node does not have a prefix then this method returns the value of the NodeName property.

See also: NamespaceURI, Prefix

NamespaceURI — read-only property

```
property NamespaceURI : DOMString
```

Returns the namespace for the current node.

Use this property to retrieve the namespace for the current node. The namespace is specified using the xmlns attribute. If the element represented by the node does not have a prefix, this method returns the default namespace that is in effect. If the element does have a prefix, this method returns the namespace associated with the prefix.

See also: LocalName, Prefix

NextSibling — read-only property

```
property NextSibling : TXpNode
```

Returns the node immediately following the current node.

Use this property to retrieve the node following the current node. If there is not a node following the current node, this property returns nil.

The following code example illustrates the use of this method:

```
aChild := aNode.FirstChild;
while (aChild <> nil) do begin
  WriteLn(Log, aChild.NodeName);
  aChild := aChild.NextSibling;
end;
```

See also: PreviousSibling

NodeID — property

```
property NodeID : Integer
```

Default: 0

Specifies a user-definable ID for the node.

This property is provided for the convenience of the developer. Use this property to assign the node a unique identifier. It could also be used to store a reference (i.e., pointer) to another set of data associated with the node.

When a node is cloned, the value of its NodeID property is copied to the clone.

See also: CloneNode

NodeName — property

```
property NodeName : DOMString
```

Default : Empty string

Returns a name for the node.

The name of a node depends upon the type of element represented by the node. Note that this property differs from the DOM specification. The DOM specification declares this property as a read-only property but XMLPartner allows the application to modify this property.

The following table lists the rules for the NodeName property:

Class	NodeName	NodeValue
TXpAttribute	Name of attribute	Value of attribute
TXpComment	#comment	Content of the comment
TXpCDATASection	#cdata-section	Content of the CDATA section
TXpDocument	#document	Empty string
TXpDocumentFragment	#document-fragment	Empty string
TXpDocumentType	Document type name	Empty string
TXpElement	Value of TagName property	Empty string
TXpEntityReference	Name of referenced entity	Empty string
TXpNotation	Name of notation	Empty string
TXpProcessingInstruction	Value of Target property	Entire content excluding the target
TXpText	#text	Content of the text node

See also: NodeType, NodeValue

```
property NodeStringType : DOMString
```

Returns the node's type as a string.

Use this property to retrieve a human-readable version of the node's type. The following table lists the values returned for each type of node:

Class	NodeStringType
TXpAttribute	ATTRIBUTE_NODE
TXpComment	COMMENT_NODE
TXpCDATASection	CDATA_SECTION_NODE
TXpDocument	DOCUMENT_NODE
TXpDocumentFragment	DOCUMENT_FRAGMENT_NODE
TXpDocumentType	DOCUMENT_TYPE_NODE
TXpElement	ELEMENT_NODE
TXpEntityReference	ENTITY_REFERENCE_NODE
TXpNotation	NOTATION_NODE
TXpProcessingInstruction	PROCESSING_INSTRUCTION_NODE
TXpText	TEXT_NODE

See also: NodeType, NodeValue

```
property NodeType : Integer
```

Returns a code indicating the node's type.

Use this method to determine the current node's type. This property is provided as an alternative to checking the node's class. The constant values returned by this property are shown in the following table:

Constant	Value	Node Type
ELEMENT_NODE	1	The node is a TXpElement.
ATTRIBUTE_NODE	2	The node is a TXpAttribute.
TEXT_NODE	3	The node is a TXpText.
CDATA_SECTION_NODE	4	The node is a TXpCDATASection.
ENTITY_REFERENCE_NODE	5	The node is a TXpEntityReference.
ENTITY_NODE	6	The node is a TXpEntity. Note that this value is not used in the object model.
PROCESSING_INSTRUCTION_NODE	7	The node is a TXpProcessingInstruction.
COMMENT_NODE	8	The node is a TXpComment.
DOCUMENT_NODE	9	The node is a TXpDocument.
DOCUMENT_TYPE_NODE	10	The node is a TXpDocumentType.
DOCUMENT_FRAGMENT_NODE	11	The node is a TXpDocumentFragment.
NOTATION_NODE	12	The node is a TXpNotation.

The following example illustrates the use of this property:

```
if aNode.NodeType = ATTRIBUTE_NODE then
  aListBox.Items.Add(aNode.NodeValue);
```

See also: NodeStringType, NodeValue

NodeValue — property

```
property NodeValue : DOMString
```

Default: Empty string

Returns the current node's value.

Use this property to retrieve the value of the node. The node value depends upon the type of node, as shown in the following table:

Class	NodeName	NodeValue
TXpAttribute	Name of attribute	Value of attribute
TXpComment	#comment	content of the comment
TXpCDATASection	#cdata-section	content of the CDATA section
TXpDocument	#document	empty string
TXpDocumentFragment	#document-fragment	empty string
TXpDocumentType	document type name	empty string
TXpElement	value of TagName property	empty string
TXpEntityReference	name of referenced entity	empty string
TXpNotation	name of notation	empty string
TXpProcessingInstruction	value of Target property	entire content excluding the target
TXpText	#text	content of the text node

See also: NodeName, NodeType

OwnerDocument — read-only property

```
property OwnerDocument : TXpDocument
```

Identifies the XML document with which this node is associated.

Use this property to reference the TXpDocument containing the node. When a node is cloned, the new node is assigned the original node's OwnerDocument. When creating a TXpNode from scratch, use the ForceOwnerDocument method to assign an owning document to the node and its child nodes.

See also: CloneNode, ForceOwnerDocument

ParentNode — read-only property

```
property ParentNode : TXpNode
```

Returns the node containing the current node.

Use this property to reference the node's parent node. If the node has just been created and not yet added to an object model, or it has been removed from an object model, this property returns nil.

Prefix — read-only property

```
property Prefix : DOMString
```

Returns the prefix of the current node's name.

When dealing with nodes whose name is qualified with a namespace (e.g., "cereal:SugarBombs"), use this property to retrieve the node's local name. In the previous example, the prefix is "cereal." If a node does not have a prefix then this method returns an empty string.

See also: LocalName, NamespaceURI

PreviousSibling — read-only property

```
property PreviousSibling : TXpNode
```

Returns the node immediately preceding the current node.

Use this property to retrieve the node preceding the current node. If there is not a node before the current node, this property returns nil.

The following code example illustrates the use of this method:

```
aChild := aNode.LastChild;
while (aChild <> nil) do begin
  WriteLn(Log, aChild.NodeName);
  aChild := aChild.PreviousSibling;
end;
```

See also: NextSibling

Release — method

```
procedure Release;
```

Decrements the node's reference count.

Nodes are reference counted. Use this method to decrement the node's reference count. Use the AddRef method to increment the node's reference count. AddRef is automatically called when the node is created or when the node is added to the document tree via AppendChild or InsertBefore.

When the reference count drops to zero, the node is freed. When you have finished adding a node to a tree, be sure to call the Release method.

The following example illustrates the use of this method:

```
{ Create a DTD. }
oDocType := aXMLObjModel.Document.CreateDocumentType('doc');
oDTDElem := oDocType.CreateDTDElement('doc', CONTENT_ELEMENTS);
oDocType.AppendChild(oDTDElem);
oDTDElem.Release;
```

See also: AddRef

RemoveAll — method

```
procedure RemoveAll;
```

Frees all child nodes of the current node.

Use this method to destroy all child nodes, and their nodes, of the current node. The memory associated with each node is freed.

See also: RemoveChild, ReplaceChild

RemoveChild — method

```
function RemoveChild(oRefChild: TXpNode): TXpNode;
```

Removes the specified node from the node's list of children.

Use this method to remove a child node from a parent node's list of children. oRefChild is the node to remove. The node oRefChild is not freed. Instead, this function returns the node instance so that it may be freed or re-used by the application. Call the removed node's Release method when the application has finished with the removed node.

See also: Release, RemoveAll, ReplaceChild

ReplaceChild — method

```
function ReplaceChild(oNewChild, oRefChild: TXpNode): TXpNode;
```

Replaces a child node with another node.

Use this method to replace a child node with another node. oNewChild is the replacement node. oRefChild is the node to be replaced. This function does not free the replaced node. Instead, it returns a reference to oRefChild so that the application may free or re-use the node. When the application has finished using oRefChild, it must call oRefChild's Release method.

The following example replaces an old element with a new element:

```
aNewElem := anObjModel.Document.CreateElement('contact');
aNewElem.SetAttribute('name', 'Joe Smith');
anOldElem := aParentNode.ReplaceChild(aNewElem, anOldElem);
aNewElem.Release;
if anOldElem <> nil then
  anOldElem.Release;
```

The following example replaces the current node with another node:

```
anOldElem := anOldElem.ParentNode.ReplaceChild(aNewElem,
anOldElem);
aNewElem.Release;
if anOldElem <> nil then
  anOldElem.Release;
```

See also: RemoveAll, RemoveChild

Supports — method

```
function Supports(sFeature, sVersion : DOMString): Boolean;
```

Identifies whether a particular feature is supported.

Use this method to determine what features a node supports. sFeature is the package name. XMLPartner supports only one package name: "XML." sVersion is the version number of the package. XMLPartner supports two versions: "1.0" and "2.0."

This function returns True if the specified feature and version is supported. Otherwise, it returns False.

Text — read-only property

```
property Text : DOMString
```

Returns the text of the current node and all of its child nodes.

Use this method to retrieve the text associated with the current node and its child nodes. This method includes text from any node that is an instance of TXpText. The text does not include tags.

See also: XMLDocument

XmlDocument — read-only property

```
property XMLDocument : DOMString
```

Returns the current node and its child nodes as a well-formed XML document.

Use this property to retrieve an in-memory version of a portion of the XML document. For very large XML documents, this may use a large amount of memory since the application will be using memory to store both the tree and the string returned by this function.

See also: Text

TXpDocument Class

The TXpDocument class implements the Document interface as defined in the DOM specification. It represents the entire XML document and usually occurs within the context of an instance of TXpObjModel. Conceptually, an instance of TXpDocument is the root of the document tree and provides the primary access to the document's data.

Note that an instance of TXpDocument is not the root element of an XML document. To access the document's root element, use the DocumentElement property.

Since objects such as elements, text nodes, comments, and processing instructions cannot exist outside the context of a TXpDocument, the TXpDocument interface contains factory methods needed to create these objects. These objects are associated with a specific instance of TXpDocument via the OwnerDocument property of the TXpNode class.

Hierarchy

TObject (VCL/CLX)

❶ TXpNode (XPDOM). 139

TXpDocument (XPDOM)

Properties

ActualCDATA
❶ Attributes
❶ ChildNodes
DocType
DocumentElement
DomImplementation
❶ FirstChild
FormattedOutput
IdAttribute
IgnoreCase
❶ LastChild
❶ LevelCode
❶ LocalName
❶ NamespaceURI
❶ NextSibling
NoCharReplacement
❶ NodeID
❶ NodeName
❶ NodeStringType
❶ NodeType
❶ NodeValue
❶ OwnerDocument
❶ ParentNode
❶ Prefix
❶ PreviousSibling
❶ Text
❶ XmlDocument
XmlDTD

Methods

❶ AddRef
❶ AppendChild
❶ CloneNode
CreateAttribute
CreateAttributeNS
CreateCDATASection
CreateComment
CreateDocumentFragment
CreateDocumentType
CreateElement
CreateElementNS
CreateEntityReference

CreateProcessingInstruction
CreateTextNode
❶ ForceOwnerDocument
❶ GetChildNodesByNodeType
GetElementsByTagName
GetElementsByTagNameNS
❶ GetNodesByNodeType
❶ HasAttributes
❶ HasChildNodes
ImportNode
❶ InsertBefore
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
❶ Supports

4

Events

OnFunction
OnFunctionAvailable
OnSystemProperty
OnVariableLookup

Reference Section

ActualCDATA — property

```
property ActualCDATA : Boolean
```

Default: False

Controls format of CDATA sections when written to file or string.

By default, CDATA sections are output with the <![CDATA[]]> tags. Set this property to True in order to output the CDATA sections as text, omitting the <![CDATA[]]> tags.

CreateAttribute — method

```
function CreateAttribute(sName : DOMString) : TXpAttribute;
```

Creates an attribute node.

Use this method to create an attribute node. sName is the name of the attribute. This function returns an instance of TXpAttribute. The attribute may be attached to an instance of TXpElement using the SetAttributeNode method of the TXpElement class.

See also: CreateAttributeNS, TXpElement.SetAttributeNode

CreateAttributeNS — method

```
function CreateAttributeNS(
  sNamespaceURI, sQualifiedName : DOMString) : TXpAttribute;
```

Creates an attribute node within a specific namespace.

Use this method to create an instance of TXpAttribute within the scope of a particular namespace. sNamespaceURI is the URI of the namespace. sQualifiedName is the node's name and may contain a local name and, optionally, a prefix. This function returns an instance of TXpAttribute.

Once the attribute has been created, it may be attached to an instance of TXpElement using the SetAttributeNode method of the TXpElement class.

See also: CreateAttribute, TXpElement.SetAttributeNode

CreateCDATASection method

```
function CreateCDATASection(
  sData : DOMString = '') : TXpCDATASection;
```

Creates a CDATA node.

Use this method to create a CDATA node. sData is an optional set of data to be stored in the CDATA section. This function returns an instance of TXpCDATASection. The CDATA node is not attached to an element.

The following example creates a CDATA section and adds it as a child of an existing node:

```
aCDATA := aObjModel.Document.CreateCDATASection('<html>');
aMemoNode.AppendChild(aCDATA);
aCDATA.Release;
```

CreateComment method

```
function CreateComment(sData : DOMString = '') : TXpComment;
```

Creates a comment node.

Use this method to create a Comment node. sData is an optional comment to be stored in the comment node. This function returns an instance of TXpComment. The following example creates a comment and inserts it as the child of a node:

```
aComment := anObjModel.Document.CreateComment(
  'This is a comment');
aNode.AppendChild(aComment);
aComment.Release;
```

CreateDocumentFragment method

```
function CreateDocumentFragment : TXpDocumentFragment;
```

Creates an empty document fragment node.

This method returns an instance of TXpDocumentFragment.

CreateDocumentType method

```
function CreateDocumentType(
  sQualified, sPublicID, sSystemID : DOMString) : TXpDocumentType;
```

Creates a document type node which is the starting point for a Document Type Definition (DTD).

Use this method as a starting point for the creation of a DTD. The DTD may be internal or external. If the DTD is internal, specify the DTD's name in sQualified and leave sPublicID and sSystemID blank. To specify an external DTD, set sQualified, sPublicID, and sSystemID as appropriate. sQualified may contain a local name and optional prefix (e.g., "accessories:zippers"). sPublicID, an optional parameter, is the DTD's public identifier. sSystemID, a required parameter for external DTDs, is the DTD's system identifier.

This function returns an instance of TXpDocumentType. After the document type has been created, it may be added to the document tree via the AppendChild or InsertBefore methods. The document type must be inserted prior to the root element of the XML document.

The following code example illustrates the creation of an internal DTD:

```
aDocType := aObjModel.Document.CreateDocumentType('doc', '', '');
aDTDElem := aDocType.CreateDTDElement('doc', CONTENT_ELEMENTS);
aDocType.AppendChild(aDTDElem);
aDTDElem.Release;
aObjModel.Document.AppendChild(aDocType);
aDocType.Release;
```

See also: TXpNode.AppendChild, DocType, TXpNode.InsertBefore

CreateElement method

```
function CreateElement(sTagName : DOMString) : TXpElement;
```

Creates an element node.

Use this method to create an element with a specific name. sTagName is the element's name and is case-sensitive. This function returns an instance of TXpElement.

The following example illustrates the creation of an element:

```
anElement := anObjModel.Document.CreateElement('document');
```

See also: CreateElementNS

CreateElementNS — method

```
function CreateElementNS(
  sNamespaceURI, sQualifiedName : DOMString) : TXpElement;
```

Creates an element node within the specified namespace.

Use this method to create an element within a specific namespace. sNamespaceURI is the URI of the namespace. sQualifiedName is the element's name and is case-sensitive. It must contain a local name. A prefix is optional. This function returns an instance of TXpElement.

The following example illustrates the creation of an element:

```
anElement := anObjModel.Document.CreateElementNS
  ('http://www.nakedmolerats.com/zoos/locations','addr:zoo');
```

See also: CreateElement

CreateEntityReference — method

```
function CreateEntityReference(
  sName : DOMString) : TXpEntityReference;
```

Creates an entity reference node.

Use this function to create a reference to an entity. sName is the name of the entity. This method returns an instance of TXpEntityReference.

CreateProcessingInstruction — method

```
function CreateProcessingInstruction(sTarget : DOMString;
  sData : DOMString = '') : TXpProcessingInstruction;
```

Creates a processing instruction node.

Use this method to create a processing instruction node. sTarget is the name of the application to receive the instruction. sData is the data associated with the processing instruction. This method returns an instance of TXpProcessingInstruction.

The following example creates a processing instruction and adds it as the child of the document:

```
aPI := aObjModel.Document.CreateProcessingInstruction(
  'xml','version="1.0"');
aObjModel.Document.AppendChild(aPI);
aPI.Release;
```

CreateTextNode — method

```
function CreateTextNode(sData : DOMString = '') : TXpText;
```

Creates a text node.

Use this method to create a text node. sData is text that is to be associated with the node. This function returns an instance of TxpText.

DocType — read-only property

```
property DocType : TXpDocType
```

Provides access to the Document Type Definition (DTD) associated with the XML document.

Use this property to access the document's DTD. If a document does not have a DTD, this property returns nil.

See also: CreateDocumentType

DocumentElement — read-only property

```
property DocumentElement : TXpElement
```

Provides access to the element that is the root of the XML document.

This property is provided as a convenience to the developer. It provides direct access to the root element of an XML document.

DomImplementation — read-only property

```
property DomImplementation : TXpDomImplementation
```

Provides access to the DOM implementation associated with this document.

```
property FormattedOutput : Boolean
```

Default: False

Indicates whether a saved XML document is to include line breaks and indents to improve readability.

Use this property to improve readability of a saved XML document. By default, XML documents are output just as they are stored with no additional line feeds or spacing. Setting this property to True inserts a carriage return and line feed after each element, comment, and processing instruction node. In addition, each nested element, processing instruction, and comment nodes is indented two spaces per level of the node.

To illustrate this property, the following XML document is read into the XML editor provided in the Examples subfolder:

```
<?XML VERSION="1.0" ENCODING="UTF-8" ?>
<!DOCTYPE TIRE []>
<TIRE MODEL="X-One" SIZE="P185/70R15">
<MANUFACTURER NAME="Michelin"
HREF="http://www.michelin.com"/><DESCRIPTION>
The all-season T speed-rated touring tire with Radial XSE
Technology to deliver the ultimate in mileage and wet traction
for today's performance touring and domestic luxury automobiles.
<!-- Available end of June --></DESCRIPTION><UTQC TREADWEAR="620"
TRACTION="A" TEMPERATURE="B" Durability="AA"/><PRICE
UNIT="Dollars">265.78 <!-- not including tax --> </PRICE></TIRE>
```

When the XML document is saved with FormattedOutput set to True, the XML document now appears as follows:

```
<?XML VERSION="1.0" ENCODING="UTF-8" ?>
<!DOCTYPE TIRE [
]>
<TIRE MODEL="X-One" SIZE="P185/70R15">
  <MANUFACTURER NAME="Michelin" HREF="http://www.michelin.com"/>
  <DESCRIPTION>The all-season T speed-rated touring tire with
Radial XSE Technology to deliver the ultimate in mileage and wet
traction for today's performance touring and domestic luxury
automobiles.<!-- Available end of June --></DESCRIPTION>
  <UTQC TREADWEAR="620" TRACTION="A" TEMPERATURE="B"
Durability="AA"/>
  <PRICE UNIT="Dollars">265.78<!-- not including tax --></PRICE>
</TIRE>
```

See also: TXpObjModel.SaveToFile

GetElementsByTagName method

```
function GetElementsByTagName(sName : DOMString) : TXpNodeList;
```

Returns all elements having a given TagName.

Use this method to find all elements in a document having a specific tag name. sName is the case-sensitive name on which the match is made. This function returns an instance of TXpNodeList containing all instances of TXpElement with a matching node name. The elements are in the same order as they would be encountered in a preorder traversal of the document tree. If sName is set to '*', all elements in the document are returned.

If no elements match sName then the returned TXpNodeList is empty. To use a case-insensitive search, set the IgnoreCase property to True. The following example illustrates the use of this method:

```
aNodeList := aXMLDoc.GetElementsByTagName('gorilla');
```

See also: GetElementsByTagNameNS, IgnoreCase, TXpElement.TagName

GetElementsByTagNameNS method

```
function GetElementsByTagNameNS(
  sNamespaceURI, sLocalName : DOMString) : TXpNodeList;
```

Returns all elements within the specified namespace having a given TagName.

Use this method to find all elements in a document residing in a specific namespace and having a specific tag name. sNamespaceURI is the namespace of the elements. sName is the case-sensitive TagName being sought. This function returns an instance of TXpNodeList containing all instances of TXpElement with a matching node name. The elements are in the same order as they would be encountered in a preorder traversal of the document tree. If sName is set to '*', all elements within the namespace are returned.

If no elements match the search parameters then the returned TXpNodeList is empty. To use a case-insensitive search, set the IgnoreCase property to True. The following example illustrates the use of this method:

```
aNodeList := aXMLDoc.GetElementsByTagNameNS(
  '../carpentry', 'tool');
```

See also: GetElementsByTagName, IgnoreCase, TXpElement.TagName

IdAttribute — read-only property

Default: “id”

Specifies the name of the XML document’s ID attribute.

Use this property to identify which attribute in the XML document is to be used as its ID attribute.

IgnoreCase — property

```
property IgnoreCase : Boolean
```

Default: False

Specifies whether searches are to be case-sensitive.

By default, all searches are case-sensitive. Use this property to enable or disable case-sensitive searches using GetElementsByTagName and GetElementsByTagNameNS. To use case-insensitive searches, set this property to True.

See also: GetElementsByTagName, GetElementsByTagNameNS

ImportNode — method

```
function ImportNode(oNode : TXpNode; bDeep : Boolean) : TXpNode;
```

Clones an existing node and places the node in the current document.

Use this method to make a copy of the node and place it within the current document. oNode is the node that is to be cloned. This function does not modify oNode in any way. To copy only oNode and its attributes, set bDeep to False. To copy oNode and all of its children, set bDeep to True.

This method returns the new copy of oNode. The returned node is attached to the document but does not have a parent node.

NoCharReplacement — property

```
property NoCharReplacement : Boolean
```

Default: False

Controls replacement of characters greater than ASCII 126 with the entity equivalent.

By default, characters greater than ASCII 126 will be replaced with their textual equivalent when outputting the XML document. For example, the character 127 is replaced with “&#127;”. To disable this behavior, set this property to True.

OnFunction — event

```
property OnFunction : TXpFunctionEvent
```

```
TXpFunctionEvent = procedure(oOwner : TObject;
  sFunctionName : DOMString; oArguments : TStringList;
  var sResult : DOMString) of object;
```

Defines an event handler called when a user function is referenced in an XPATH string.

The OnFunction event should not be used at this level. Instead, use the OnFunction event of the TXpObjModel class.

See also: TXpObjModel.OnFunction, TXpObjModel.OnFunctionAvailable

OnFunctionAvailable — event

```
property OnFunctionAvailable : TXpFunctionAvailableEvent
```

```
TXpFunctionAvailableEvent = procedure(oOwner : TObject;
  sFunctionName : DOMString; var bAvailable : Boolean);
```

Defines an event that is called when the standard XPath function named “function-available” is encountered.

The OnFunctionAvailable event should not be used at this level. Instead, use the OnFunctionAvailable event of the TXpObjModel class.

See also: TXpObjModel.OnFunction, TXpObjModel.OnFunctionAvailable

OnSystemProperty — event

```
property OnSystemProperty : TXpVariableLookupEvent
```

```
TXpVariableLookupEvent = procedure(oOwner : TObject;
  sVarName : DOMString; var sVarValue : DOMString) of object;
```

Declares an event handler that is called when the standard XSLT system-property function is encountered.

The OnSystemProperty event should not be used at this level. Instead, use the OnSystemProperty event of the TXpObjModel class.

See also: TXpObjModel.OnSystemProperty

OnVariableLookup — event

```
property OnVariableLookup : TXpVariableLookupEvent
```

```
TXpVariableLookupEvent = procedure(oOwner : TObject;
  sVarName : DOMString; var sVarValue : DOMString) of object;
```

Defines an event that is called when a variable is referenced in an XPath expression.

The OnVariableLookup event should not be used at this level. Instead, use the OnVariableLookup event of the TXpObjModel class.

See also: TXpObjModel.OnVariableLookup

XmlDTD — read-only property

```
property XmlDTD : DOMString
```

Returns the document's Document Type Definition (DTD) as a formatted document.

Use this method to retrieve a copy of the DTD in string format.

TXpNodeList Class

The TXpNodeList class implements the NodeList interface defined in the DOM specification and is used to contain a collection of nodes. The ChildNodes property of the TXpNode class stores child nodes in an instance of this class. When searching for nodes using the SelectNodes method of the TXpElement class, the result of the search is returned in an instance of TXpNodeList.

This class provides methods that add nodes to the list, delete nodes, move nodes, and swap nodes. The CopyList method provides a way to clone an existing node list.

To directly access nodes based upon their position in the list, use the Length property to determine the number of nodes in the list. Use the Items method to access a specific node. The node list is base zero meaning that the first node is at position zero.

Hierarchy

TObject (VCL/CLX)

TXpNodeList (XPDOM)

Properties

Length	XMLDocument

Methods

Add	Exchange	Move
CopyList	IndexOf	Replace
Delete	Insert	Sort
Empty	Item	

Add — method

```
procedure Add(oNode : TXpNode);
```

Adds a node to the list.

Use this method to add a node to the end of a node list. oNode is the node to add. The following code example illustrates this method:

```
aParentNode.ChildNodes.Add(aNewNode);
```

See also: Delete, Insert

CopyList — method

```
procedure CopyList(oNodeList : TXpNodeList);
```

Adds the contents of a node list into the current node list.

Use this method to add nodes in one list to another list. The nodes are not removed from the original list. oNodeList is the node list to be copied.

Delete — method

```
procedure Delete(wIndex : Integer);
```

Deletes a node at the specified index.

Use this method to remove a node from the node list. wIndex is the position of the node to remove and is zero-based. This method calls the node's Release method. The following lines show how this method may be used:

```
aNode := aNodeList.Item(0);
aNodeList.Delete(0);
```

See also: Empty

Empty — method

```
procedure Empty;
```

Removes all nodes from the node list.

Use this method to remove all nodes from the node list. This method calls the Release method of each node in the list. The following example illustrates the use of this method:

```
if aNodeList.GetLength > 0 then
  aNodeList.Empty;
```

See also: Delete

Exchange — method

```
function Exchange(wSrc, wDest : Integer) : Boolean;
```

Switches the positions of two nodes in the node list.

Use this method to swap two nodes within the node list. wSrc is the index of the first node. wDest is the index of the second node. Both wSrc and wDest are zero-based, so the first position of an item in the list is 0, the second item is at position 1, and so on.

The following example shows how this method may be used:

```
{ Swap the first and last nodes. }
aNodeList.Exchange(0, pred(aNodeList.Length));
```

See also: Move, Replace

IndexOf — method

```
function IndexOf(oNode : TXpode) : Integer;
```

Returns the index of the specified node within the node list.

Use this method to find the position of the node within the node list. oNode is a reference to the node. If the node is in the list, a zero-based position is returned. If the node is not in the list, then –1 is returned. The following example shows how to use this method:

```
aPos := aNodeList.IndexOf(aNode);
if aPos >= 0 then
  aNodeList.Delete(aPos);
```

Insert — method

```
procedure Insert(wIndex : Integer; oNode : TXpNode);
```

Inserts a node into the node list at the specified position.

Use this method to insert a node into the node list. wIndex is the zero-based position into which the node is inserted. The items at that position, and the subsequent items, are shifted up. oNode is the node to insert into the list. This method allows the same node to be inserted into the list more than once. When the node is inserted, this method calls the node's AddRef method.

The following example illustrates the use of this method:

```
{ Insert the node at the beginning of the list. }
aNodeList.Insert(0, aNode);
```

See also: Add

Item — method

```
function Item(wIndex : Integer) : TXpNode;
```

Returns the node at the specified position.

Use this method to retrieve a specific node within the node list. wIndex is the zero-based position of the node to retrieve. If a node does not exist at the specified position, nil is returned. The following example illustrates the use of this method:

```
aNode := aNodeList.Item(0);
aNodeList.Delete(0);
```

See also: Length

Length — read-only property

```
property Length : Integer
```

Returns the number of nodes in the node list.

Use this property to determine the number of nodes in the node list. If the node list is empty, this property returns zero.

Move — method

```
function Move(wSrc, wDest : Integer) : Boolean;
```

Moves a node from one position to another within the node list.

Use this method to move a node from one position to another within the node list. wSrc is the zero-based position of the node to move. wDest is the new, zero-based position of the node. This method returns True if the move was successful. Otherwise, it returns False.

The following example illustrates the use of this method:

```
{ Move the first node to the last position. }
aNodeList.Move(0, pred(aNodeList.Length));
```

See also: Exchange, Replace

Replace — method

```
procedure Replace(wIndex : Integer; oNode : TXpNode);
```

Replaces the node at the specified position with another node.

Use this method to replace a node in the node list with another node. wIndex is the zero-based position of the node to be replaced. oNode is the replacement node. This method calls the Release method of the node being replaced.

See also: Exchange, Move

Sort — method

```
procedure Sort(sAttribute : DOMString = ''; wOrder : Integer = 0);
```

Sorts the node list according to a specific attribute or to the node text.

Use this method to sort the nodes within a node list according to the values of a specific attribute or the node text. sAttribute is the name of the attribute on which the nodes are to be sorted. If sAttribute is an empty string then the nodes are sorted by their Text property.

wOrder specifies whether the nodes are to be sorted in ascending or descending order. To sort in ascending order, set wOrder to zero. To sort in descending order, set wOrder to one. If a value other than zero or one is specified, the nodes are sorted in ascending order.

When sorting by attribute, if a node does not have the attribute then it is sorted as though the attribute value is an empty string.

The following example sorts the nodes according to their text attribute in descending order:

```
aNodeList.Sort('', 1);
```

The following example sorts the nodes in ascending order according to their name attribute:

```
aNodeList.Sort('name', 0);
```

XMLDocument — property

```
property XMLDocument : DOMString
```

Returns all items in the list as an XML fragment.

TXpNamedNodeMap Class

The TXpNamedNodeMap class implements the NamedNodeMap interface in the DOM specification and is used to represent collections of nodes accessible via the node name. Note that although the TXpNamedNodeMap class implements nearly the same interface as class TXpNodeList, it does not inherit from TXpNodeList.

Use the GetNamedItem to retrieve an element via its name. Use the SetNamedItem to add an element by its name. Use the RemoveNamedItem to remove an element based upon its name.

To sequentially access the nodes within the map, use the Length property to determine the number of nodes in the map. Use the Item method to access a specific node. The Item method is base zero (i.e., the first node in the list is at position zero).

Hierarchy

TObject (VCL/CLX)

TXpNamedNodeMap (XPDOM)

Properties

Length

Methods

Add	IndexOf	Replace
Delete	Insert	SetNamedItem
Empty	Item	
GetNamedItem	RemoveNamedItem	

Add — method

```
procedure Add(oNode : TXpNode);
```

Adds a node to the node map.

Use this method to add a node to the end of a node map. oNode is the node to add. The following code example illustrates this method:

```
aNodeMap.Add(aNewNode);
```

This method adds the node even if a node bearing the same nodename is already in the map. To prevent nodes having the same name from entering the map, use the SetNamedItem method.

See also: Delete, Insert, SetNamedItem

Delete — method

```
procedure Delete(wIndex : Integer);
```

Deletes a node at the specified index.

Use this method to remove a node from the node map. wIndex is the position of the node to remove and is zero-based. This method calls the node's Release method. The following lines of code show how this method may be used:

```
aNode := aNodeMap.Item(0);
aNodeMap.Delete(0);
```

See also: Empty

Empty — method

```
procedure Empty;
```

Removes all nodes from the node map.

Use this method to remove all nodes from the node map. This method calls the Release method of each node in the map. The following example illustrates the use of this method:

```
if aNodeMap.GetLength > 0 then
  aNodeMap.Empty;
```

See also: Delete

GetNamedItem — method

```
function GetNamedItem(sName : DOMString) : TXpNode;
```

Retrieves a node by name from the node map.

Use this method to find a node by its name in the node map. sName is the name of the node to find. The map compares sName against each node's NodeName property until a matching node is found. The comparison is case-sensitive unless the node within the map has an owning document and the document's IgnoreCase property is set to True.

See also: TXpDocument.IgnoreCase, TXpNode.OwnerDocument, RemoveNamedItem

IndexOf — method

```
function IndexOf(oNode : TXpode) : Integer;
```

Returns the index of the specified node within the node map.

Use this method to find the position of the node within the node map. oNode is a reference to the node. If the node is in the map, a zero-based position is returned. If the node is not in the map then –1 is returned. The following example shows how to use this method:

```
aPos := aNodeMap.IndexOf(aNode);
if aPos >= 0 then
  aNodeMap.Delete(aPos);
```

Insert — method

```
procedure Insert(wIndex : Integer; oNode : TXpNode);
```

Inserts a node into the node map at the specified position.

Use this method to insert a node into the node map. wIndex is the zero-based position into which the node is inserted. The items at that position, and the subsequent items, are shifted up. oNode is the node to insert into the map. This method allows the same node to be inserted into the map more than once. When the node is inserted, this method calls the node's AddRef method.

The following example illustrates the use of this method:

```
{ Insert the node at the beginning of the map. }
aNodeMap.Insert(0, aNode);
```

See also: Add

Item — method

```
function Item(wIndex : Integer) : TXpNode;
```

Returns the node at the specified position.

Use this method to retrieve a specific node within the node map. wIndex is the zero-based position of the node to retrieve. If a node does not exist at the specified position then nil is returned. The following example illustrates the use of this method:

```
aNode := aNodeMap.Item(0);
aNodeMap.Delete(0);
```

See also: Length

Length — read-only property

```
property Length : Integer
```

Returns the number of nodes in the node map.

Use this property to determine the number of nodes in the node map. If the node map is empty, this property returns zero.

RemoveNamedItem — method

```
function RemoveNamedItem(sName : DOMString) : TXpNode;
```

Removes the first node from the map having the specified name.

Use this method to remove a node from the node map, based upon the node's name. sName is the name of the node to be removed. The comparison of sName to the node names is case-sensitive unless a node in the map has an owning document and the document's IgnoreCase property is set to True. If the node map contains more than one node with NodeName equal to sName, only the first node encountered is removed.

See also: GetNamedItem

Replace — method

```
procedure Replace(wIndex : Integer; oNode : TXpNode);
```

Replaces the node at the specified position with another node.

Use this method to replace a node in the node map with another node. wIndex is the zero-based position of the node to be replaced. oNode is the replacement node. This method calls the Release method of the node being replaced.

SetNamedItem method

```
function SetNamedItem(oNode : TXpNode) : TXpNode;
```

Adds a node to the node map, replacing any node already in the map having the same name.

Use this method to add a node to the node map. It is different from the Add method in that it first searches for a node in the map that contains the same nodename. If more than one node in the map bears the same name, only the first node encountered is removed.

See also: Add

TXpCharacterData Class

The TXpCharacterData class implements the CharacterData interface defined in the DOM specification. Instances of this class do not appear in an object model. Instead, it is used as a foundation for other interfaces such as TXpComment and TXpText.

This class defines several methods for setting, retrieving, deleting, and modifying character data. All offsets passed to these methods are base zero.

Hierarchy

TObject (VCL/CLX)

❶ TXpNode (XPDOM). 139

TXpCharacterData (XPDOM)

Properties

❶ Attributes
❶ ChildNodes
Data
❶ FirstChild
❶ LastChild
Length
❶ LevelCode
❶ LocalName
❶ NamespaceURI
❶ NextSibling
❶ NodeID
❶ NodeName
❶ NodeStringType
❶ NodeType
❶ NodeValue
❶ OwnerDocument
❶ ParentNode
❶ Prefix
❶ PreviousSibling
❶ Text
❶ XmlDocument

Methods

❶ AddRef
❶ AppendChild
AppendData
❶ CloneNode
DeleteData
❶ ForceOwnerDocument
❶ GetChildNodesByNodeType
❶ GetNodesByNodeType
❶ HasAttributes
❶ HasChildNodes
❶ InsertBefore
InsertData
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
ReplaceData
SubstringData
❶ Supports

AppendData — method

```
procedure AppendData(sData : DOMString);
```

Appends data to a node's existing data.

Use this method to add data to a node's existing data. sData contains the additional data. The contents of sData are added as is. This method does not insert a space between the existing data and the data specified by sData.

The following example illustrates the use of this method:

```
while not EOF(aFile) do begin
  ReadLn(aFile, someText);
  aNode.AppendData(someText);
end;
```

See also: Data, DeleteData, InsertData, ReplaceData, SubstringData

Data — property

```
property Data : DOMString
```

Default: Empty string

Provides access to the node's character data.

Use this method to get or set the node's character data.

See also: AppendData, DeleteData, InsertData, Length, ReplaceData, SubstringData

DeleteData — method

```
procedure DeleteData(wOffset, wCount : Integer);
```

Removes characters from the character data.

Use this method to remove characters from the node's Data property. wOffset is the zero-based starting position for the deletion of characters. wCount is the number of characters to delete.

If wCount is greater than the length of the data after wOffset, all characters from wOffset to the end of the data are deleted. If wOffset is larger than the number of characters then no characters are deleted.

The following code example deletes ten characters starting at the very start of the data:

```
aNode.DeleteData(0, 10);
```

See also: AppendData, Data, InsertData, ReplaceData, SubstringData

InsertData — method

```
procedure InsertData(wOffset : Integer; sData : DOMString);
```

Inserts characters at a specified position within the node's Data.

Use this method to insert characters into a node's existing data. wOffset is the zero-based position where the characters are to be inserted. sData contains the characters to be inserted. All existing characters starting at position wOffset are shifted up to make room for the new characters.

The following example inserts a string before the 26th character:

```
aNode.InsertData(25, aWAVSample);
```

See also: AppendData, Data, DeleteData, ReplaceData, SubstringData

Length — read-only property

```
property Length : Integer
```

Returns the length of the node's character data.

Use this property to determine the length of the data already in the node. If the node has no data, this property returns zero.

See also: Data

ReplaceData — method

```
procedure ReplaceData(
  wOffset, wCount : Integer; sData : DOMString);
```

Replaces existing characters within Data with another set of characters.

Use this method to replace data in the node. wOffset is the zero-based offset at which characters are to be replaced. wCount is the number of characters to replace. sData contains the replacement characters.

The number of characters in sData does not have to match the number of characters specified by wCount.

The following example replaces the first fifty characters in the node with a linefeed:

```
aNode.ReplaceData(0, 50, #10);
```

See also: AppendData, Data, DeleteData, InsertData, SubstringData

SubstringData — method

```
function SubStringData(wOffset, wCount : Integer) : DOMString;
```

Extracts a range of characters from the node's Data property.

Use this method to extract characters from the node's Data property. wOffset is the zero-based position at which characters are to be extracted. wCount is the number of characters to extract. If wCount is greater than the number of characters after position wOffset, this method returns all the characters from wOffset to the end of the data. If wOffset is greater than the number of characters in the node, an empty string is returned.

See also: AppendData, Data, DeleteData, InsertData, ReplaceData

TXpAttribute Class

The TXpAttribute class implements the Attr interface as defined in the DOM specification. An instance of this class must have an instance of class TXpElement as its parent node. The allowed values for an attribute are typically defined in the Document Type Definition (DTD). In XMLPartner, the DTD is implemented via the TXpDocumentType class in Chapter 5.

Attribute objects are not considered part of the document tree, therefore the ParentNode, PreviousSibling, and NextSibling methods of this class return nil. The DOM specification takes the view that attributes are properties of elements and do not have an identity apart from the elements with which they are associated.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 139
  - TXpAttribute (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- OwnerElement
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- Specified
- ❶ Text
- Value
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

Reference Section

Name — property

```
property Name : DOMString
```

Returns the name of the attribute.

Use this property to retrieve the name of the attribute. The name may be qualified, containing a prefix and a local name. To obtain the prefix or local name without parsing the Name property, use the Prefix and LocalName properties.

See also: TXpNode.LocalName, TXpNode.Prefix

OwnerElement — read-only property

```
property OwnerElement : TXpElement
```

Identifies the element that owns the attribute.

Use this method to obtain the element with which the attribute is associated.

Specified — property

```
property Specified : Boolean
```

Default: True

Indicates whether the attribute was specified with the element or with the element declaration.

If the attribute was associated with the element via the element declaration in the Document Type Definition (DTD), this property returns False. Otherwise, this property returns True.

Value — property

```
property Value : DOMString
```

Default: Empty string

Returns the value of the attribute.

Use this property to set or obtain the attribute's value.

TXpElement Class

The TXpElement class implements the Element interface defined in the DOM specification. An element is the most commonly found object in the document tree. Elements may be nested. Elements may have children that are instances of TXpCDATASection, TXpComment, and TXpText.

Elements may have one or more attributes, available through the Attributes property of class TXpNode or the GetAttribute and GetAttributeNode methods of this class. If a value is not explicitly supplied for an attribute but the attribute's declaration contains a default value, any attempt to retrieve the attribute's value from the element will return the attribute's default value.

The TXpElement class supports the querying of nodes using the XPath 1.0 recommendation. Use the SelectNodes method to retrieve a list of nodes matching an expression. Use the SelectBoolean, SelectInteger, SelectNumber, SelectSingleNode, or SelectString methods to retrieve a value based upon an expression.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 139
  - TXpElement (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ElementText
- ❶ FirstChild
- FullEndTag
- IgnoreEndTag
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- TagName
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- CloneNode
- CreateChildCDataSection
- CreateChildElement
- CreateChildText
- FindElement
- ❶ ForceOwnerDocument
- GetAttribute
- GetAttributeNode
- GetAttributeNodeNS
- GetAttributeNS

GetChildElementsByTagName
❶ GetChildNodesByNodeType
GetElementsByTagName
GetElementsByTagNameNS
GetElementsByTagNameWith-Attribute
❶ GetNodesByNodeType
HasAttribute
❶ HasChildNodes
❶ InsertBefores
❶ IsAfter
Normalize
❶ Release
❶ RemoveAll
RemoveAttribute
RemoveAttributeNode
RemoveAttributeNS
❶ RemoveChild
❶ ReplaceChild
SelectBoolean
SelectInteger
SelectNodes
SelectNumber
SelectSingleNode
SelectString
SetAttribute
SetAttributeNode
SetAttributeNodeNS
SetAttributeNS
❶ Supports

CreateChildCDataSection — method

```
function CreateChildCDataSection(
  sText : DOMString) : TXpCDataSection;
```

Creates a CDATA node.

Use this method to create a CDATA node containing the specified text. sText is the text to include in the node. This method returns an instance of TXpCDataSection and automatically makes the new node a child of the current element.

CreateChildElement — method

```
function CreateChildElement(sElem : DOMString) : TXpElement;
```

Creates an element node.

Use this method to create an element node. sElem is the name of the new element. This method returns an instance of TXpElement and automatically makes the new element a child of the current element.

CreateChildText — method

```
function CreateChildText(sText : DOMString) : TXpText;
```

Creates a text node.

Use this method to create a text node. sText is the text to be associated with the new node. This method returns an instance of TXpText and automatically makes the new text node a child of the current element. The following example creates a text block as a child of an existing node:

```
aMemoNode.CreateChildText(
 'The train is booked solid for Tuesday.');
```

ElementText — read-only property

```
property ElementText : DOMString
```

Returns the text associated with the element.

This method returns a DOMString containing all of the text from each of its child nodes that are text nodes (i.e., an instance of TXpText). A space character separates the text from each node. For example, an element has four child nodes, two of which are text nodes. The first text node contains the text “abra” and the second text node contains the text “cadabra.” In this case, the ElementText property returns the value “abra cadabra.”

FindElement — method

```
function FindElement(sName : DOMString) : TXpElement;
```

Searches for a child element having the specified name.

Use this method to search the current element's child nodes, and their children, for an element having a specific name. sName contains the element name to find. The search is case-sensitive unless the element has an owner document and the document's IgnoreCase property is True. The search is conducted in a preorder traversal (i.e., left to right).

The following example loads a document definition from memory. It then searches for the element "product:surveys." If the element is not found, it creates it under the root element:

```
var
  aRoot, aElem: TXmlElement;
begin
  aObjModel.LoadMemory(sDefinition[1]);
  aRoot :=
aObjModel.Document.DocumentElement.FindElement('product:surveys'
);
  if aRoot = nil then begin
    aRoot :=
aObjModel.Document.DocumentElement.CreateChildElement
              ('product:surveys');
    aElem := aObjModel.Document.CreateElement('product:survey-
file');
    aElem.SetAttribute('file', MainForm.ProjectFile);
    aRoot.AppendChild(aElem);
    aElem.Release;
    aObjModel.SaveToFile('e:\prodsrvy\definition.xml');
end;
```

See also: TXpDocument.IgnoreCase

FullEndTag — property

```
property FullEndTag : Boolean
```

Default: False

Forces the element to output a full end tag even if it has no children.

By default, when an element is output to file or memory and the element has no children then it is given an empty-element tag (i.e., "<anElement />"). To force the output of a full end tag (i.e., "<anElement></anElement>"), set this property to True.

See also: IgnoreEndTag

GetAttribute — method

```
function GetAttribute(sName : DOMString) : DOMString;
```

Retrieves an attribute value where the attribute has a specific name.

Use this function to retrieve the value of a named attribute from the element. sName is the name of the attribute. The comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True. This method returns the value of the attribute's NodeValue property. If a matching attribute is not found, this method returns an empty string.

To find an attribute within a particular namespace, use the GetAttributeNS method.

See also: GetAttributeNode, GetAttributeNS, HasAttributes, TXpDocument.IgnoreCase, TXpNode.NodeValue

GetAttributeNode — method

```
function GetAttributeNode(sName : DOMString) : TXpAttribute;
```

Retrieves an attribute node from the element, given a specific attribute name.

Use this function to retrieve an attribute node having a specific name. sName is the name of the attribute. The name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True. This method returns an instance of TXpAttribute having the specific name. If a matching attribute is not found, this method returns the value nil.

To find an attribute within a particular namespace, use the GetAttributeNodeNS method.

See also: GetAttribute, GetAttributeNodeNS, HasAttributes, TXpDocument.IgnoreCase

GetAttributeNodeNS method

```
function GetAttributeNodeNS
  (sNamespaceURI, sLocalName : DOMString) : TXpAttribute;
```

Retrieves an attribute within a specified namespace and having a specific name.

Use this method to retrieve an attribute node from the element, where the attribute is within a specific namespace and has a specific name. sNamespaceURI is the namespace in which the attribute resides. sLocalName is the local name of the attribute. The namespace and name comparisons are case-sensitive.

This method returns an instance of TXpAttribute. If a matching attribute is not found, this method returns the value nil.

To find an attribute regardless of namespace, use the GetAttributeNode method.

See also: GetAttributeNode, HasAttributes

GetAttributeNS method

```
function GetAttributeNS(
  sNamespaceURI, sLocalName : DOMString) : DOMString;
```

Retrieves an attribute value within a specified namespace and having a specific name.

Use this method to retrieve an attribute value where the attribute resides in a specific namespace and has a specific name. sNamespaceURI is the namespace in which the attribute resides. sLocalName is the local name of the attribute. The namespace and name comparisons are case-sensitive.

This method returns the value of the attributes NodeValue property. If a matching attribute is not found, this method returns an empty string.

To find an attribute value regardless of namespace, use the GetAttribute method.

See also: GetAttribute, HasAttributes, TXpNode.NodeValue

GetChildElementsByTagName method

```
function GetChildElementsByTagName(
  sName : DOMString) : TXpNodeList;
```

Retrieves all immediate child elements having the specified tag name.

Use this method to retrieve all immediate children of the element having a specific TagName. sName is the tag name sought. The tag name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True. To retrieve all elements regardless of TagName, specify the value '*' (i.e., an asterisk) for sName.

This method returns an instance of TXpNodeList. If none of the immediate child elements matched the specified tag name, the node list is empty. Otherwise, the node list contains one or more elements ordered as they would be encountered if walking through the child nodes from left to right.

To perform a deep search of the entire node tree below the element, use the GetElementsByTagName or GetElementsByTagNameNS method.

See also: GetElementsByTagName, GetElementsByTagNameNS, TXpDocument.IgnoreCase , TagName

GetElementsByTagName method

```
function GetElementsByTagName(sName : DOMString) : TXpNodeList;
```

Retrieves all descendant elements having the specified tag name.

Use this method to perform a recursive search for all child elements having a specific TagName. sName is the tag name sought. The tag name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True.

This method returns an instance of TXpNodeList. If none of the child elements match the specified tag name, the node list is empty. Otherwise, the elements in the node list are ordered as though a preorder traversal were performed.

To perform a search of the element's immediate children only, use the GetChildElementsByTagName method.

See also: GetChildElementsByTagName, GetElementsByTagNameNS, GetElementsByTagNameWithAttribute, TXpDocument.IgnoreCase, TagName

GetElementsByTagNameNS — method

```
function GetElementsByTagNameNS(
  sNamespaceURI, sLocalName : DOMString) : TXpNodeList;
```

Retrieves all descendant elements within a specified namespace and having a specific tag name.

Use this method to perform a recursive search for all child elements within a specific namespace and having a specific LocalName. sNamespaceURI is the namespace in which the elements must reside. sLocalName is the local name being sought. The namespace and local name comparisons are case-sensitive.

This method returns an instance of TXpNodeList. If none of the child elements match the namespace and local name, the node list is empty. Otherwise, the elements in the node list are ordered as though a preorder traversal were performed.

See also: GetElementsByTagName, GetElementsByTagNameWithAttribute

GetElementsByTagNameWithAttribute — method

```
function GetElementsByTagNameWithAttribute(
  sName, sAttr, sValue : DOMString) : TXpNodeList;
```

Retrieves all descendant elements having the specified TagName and an attribute of the specified name and value.

Use this method to perform a recursive search for all child elements having a specific tag name and bearing an attribute having a specific name and value. sName is the TagName being sought. sAttr is the name of the attribute and sValue is the value the attribute must have. The name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is set to True. To find elements regardless of name, specify the value '*' (i.e., an asterisk) for sName.

This method returns an instance of TXpNodeList. If none of the elements matched the parameters, the list is empty. Otherwise, the list contains one or more elements ordered as though a preorder traversal were performed.

See also: GetElementsByTagName, GetElementsByTagNameNS, TXpDocument.IgnoreCase

HasAttribute — method

```
function HasAttribute(sName : DOMString) : Boolean;
```

Determines if the element has a specific attribute.

Use this method to determine if the element has an attribute with a specific name. sName is the attribute's name. The name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True. If a matching attribute is found, this method returns True. Otherwise, it returns False.

See also: GetAttribute, GetAttributeNode, GetAttributeNodeNS, GetAttributeNS

IgnoreEndTag — property

```
property IgnoreTag : Boolean
```

Default: False

Forces the output of an XML document to exclude end tags for elements lacking child nodes.

When outputting an XML document, use this property to exclude end tags for elements lacking child nodes. To exclude end tags, set this property to True. For example, the tag "<BR/>" would be written as "
." This feature is useful for HTML browser compatibility.

See also: FullEndTag

Normalize — method

```
procedure Normalize(bAddSpace : Boolean);
```

Condenses adjacent text nodes into one text node.

Use this method to collapse sibling text nodes into individual text nodes. For example, if the element has 4 child nodes with the first 3 nodes being text nodes and the last node being a comment node then this method collapses the first 3 text nodes into a single text node.

To separate the text from each node by a space character, set bAddSpace to True. Otherwise, set it to False. This method performs a full depth search for adjacent text nodes.

RemoveAttribute method

```
procedure RemoveAttribute(sName : DOMString);
```

Removes a named attribute from the element's list of attributes.

Use this method to remove a specific attribute from an element's list of attributes based upon the attribute's name. sName is the name of the attribute. The name comparison is case-sensitive unless the element has an owner document and the owning document's IgnoreCase property is True.

See also: RemoveAttributeNode, RemoveAttributeNS

RemoveAttributeNode method

```
function RemoveAttributeNode(
  oOldAttr : TXpAttribute) : TXpAttribute;
```

Removes a specific attribute from the element's list of attributes.

Use this method to remove a specific attribute from an element's list of attributes. oOldAttr is the attribute to remove. If the instance referenced by oOldAttr is not an attribute of the element then this method does nothing and returns nil. Otherwise, this function returns a reference to oOldAttr.

RemoveAttributeNS method

```
procedure RemoveAttributeNS(sNamespaceURI, sLocalName : DOMString);
```

Removes a named attribute from the element's list of attributes.

Use this method to remove a named attribute residing within a specific namespace. This method searches for the attribute within the element's list of attributes. sNamespaceURI is the attribute's namespace. sLocalName is the attribute's local name, excluding a prefix.

SelectBoolean method

```
function SelectBoolean(sQuery : DOMString) : Boolean;
```

Evaluates an XPath expression to True or False.

Use this method to return a boolean result for a specific XPath expression. sQuery is an expression conforming to the XPath 1.0 recommendation. See the SelectNodes method for example expressions.

The value returned by this method depends upon the type of value returned by the expression. The logic for determining the return value is as shown in the following table:

Expression return type	Logic
List of nodes	Returns True if one or more nodes that match the expression.
Boolean	Returns True if the evaluated expression results in the value True.
Number	Returns True if the evaluated expression results in a value other than zero.
String	Returns True if the string is one or more characters in length.

See also: SelectInteger, SelectNodes, SelectNumber, SelectSingleNode, SelectString

```
function SelectInteger(sQuery : DOMString) : Integer;
```

Evaluates an XPath expression, returning the expression's result as an integer.

Use this method to obtain an integer result for a specific XPath expression. sQuery is an expression conforming to the XPath 1.0 recommendation. See the SelectNodes method for example expressions.

The value returned by this method depends upon the type of value returned by the expression. The logic for determining the return value is as shown in the following table:

Expression return type	Logic
List of nodes	If the first node in the list is an attribute, it returns the attribute's value converted to an integer. Otherwise, this method returns the first node's text converted to an integer. If the list has no nodes, this method returns zero.
Boolean	If the expression evaluates to True, this method returns the value 1. Otherwise, this method returns the value zero.
Number	Returns the number.
String	Returns the string converted to an integer.

See also: SelectBoolean, SelectNodes, SelectNumber, SelectSingleNode, SelectString

```
function SelectNodes(sQuery : DOMString) : TXpNodeList;
```

Selects a set of nodes based upon an XPath expression.

Use this method to select a set of nodes. sQuery is an expression conforming to the XPath 1.0 recommendation. The starting node of the query is the element node for which this method is invoked. Examples of queries are as follows:

Expression	Meaning
ancestor::department/group	Finds all "group" elements directly underneath an ancestor "department" element
../employees	Finds all sibling "employee" elements.
author/first-name	Finds all "first-name" elements under child "author" elements.
..//heading	Finds all descendant "heading" elements.
/company//departments	Finds all descendant "department" elements under "company" elements from the root of the document tree.
names[registered]	Finds all child "names" elements that have a child "registered" node.
list[@ordered='yes']/item[first-of-type()]	Finds the first "item" element at its level underneath child "list" elements that have an attribute "ordered" set to "yes".

The following example searches for “sst:item” nodes below the root node whose “name” attribute is equal to “application.x.” If the node does not exist, it is created:

```
var
  aBaseElem, aElem: TXmlElement;
  aList: TXmlNodeList;
begin
  { Obtain a reference to the XML document. }
  aBaseElem := aObjModel.DocumentElement;

  aList := aBaseElem.SelectNodes(
    'sst:item[@name = "application.x"]');
  { Matching elements found? }
  if aList.Length > 0 then
    { Yes. Get the first matching element. }
    aElem := TXmlElement(aList.Item(0))
  else
  begin
    { No. Create a matching element from scratch. }
    aElem := aBaseElem.CreateChildElement('sst:item');
    aElem.SetAttribute('name', 'application.x');
  end;
  { Set the element's Value attribute. }
  aElem.SetAttribute('value', ‘100’);
  aList.Free;
end;
```

See also: SelectBoolean, SelectInteger, SelectNumber, SelectSingleNode, SelectString

SelectNumber method

```
function SelectNumber(sQuery : DOMString) : Double;
```

Evaluates an XPath expression, returning the expression's result as a double.

Use this method to obtain a numeric result for a specific XPath expression. sQuery is an expression conforming to the XPath 1.0 recommendation. See the SelectNodes method for example expressions.

The value returned by this method depends upon the type of value returned by the expression. The logic for determining the return value is as shown in the following table:

Expression return type	Logic
List of nodes	If the first node in the list is an attribute, it returns the attribute's value converted to type Double. Otherwise, this method returns the first node's text converted to type Double. If the list has no nodes, this method returns zero.
Boolean	If the expression evaluates to True, this method returns the number 1. Otherwise, it returns zero.
Number	Returns the number converted to type Double.
String	Returns the string converted to type Double.

See also: SelectBoolean, SelectInteger, SelectNodes, SelectSingleNode, SelectString

SelectSingleNode method

```
function SelectSingleNode(sQuery : DOMString) : TXpElement;
```

Returns the first element matching an XPath expression.

Use this method to obtain the first element in the list of elements matching an XPath expression. sQuery is an expression conforming to the XPath 1.0 recommendation. See the SelectNodes method for example expressions.

If the evaluated expression returns a list of one or more elements, this method returns the first element in the list. Otherwise, it returns nil.

See also: SelectBoolean, SelectInteger, SelectNodes, SelectNumber, SelectString

SelectString method

```
function SelectString(sQuery : DOMString) : DOMString;
```

Evaluates an XPath expression, returning the expression's result as a string.

Use this method to obtain a string result for a specific XPath expression. sQuery is an expression conforming to the XPath 1.0 recommendation. See the SelectNodes method for example expressions.

The value returned by this method depends upon the type of value returned by the expression. The logic for determining the return value is as shown in the following table:

Expression return type	Logic
List of nodes	If the first node in the list is an attribute, it returns the attribute's value. Otherwise, this method returns the first node's text. If the list has no nodes, this method returns an empty string.
Boolean	Returns the boolean value converted to a string.
Number	Returns the number converted to a string.
String	Returns the string as is.

See also: SelectBoolean, SelectInteger, SelectNodes, SelectNumber, SelectSingleNode

SetAttribute method

```
procedure SetAttribute(sName, sValue : DOMString);
```

Creates a new attribute and adds it to the element.

Use this method to add a named attribute to an element. sName is the name of the attribute. It may contain a qualified name (i.e., prefix and local name). sValue is the value of the attribute. If the element already contains an attribute having the same name then this method removes that attribute before adding the new attribute. Note that the search for an existing attribute does not take namespaces into account. Use the SetAttributeNS method if namespaces must be taken into account.

To declare a namespace prefix for an element, set sName to the value "xmlns:prefix" where "prefix" is the prefix for the namespace. Set sValue to the Unique Resource Identifier (URI) of the namespace. This method adds the namespace to the element's list of namespaces.

To declare the default namespace for the element, set sName to the value "xmlns" and sValue to the namespace URI.

See also: SetAttributeNode, SetAttributeNodeNS, SetAttributeNS

```
function SetAttributeNode(oNewAttr : TXpAttribute) : TXpAttribute;
```

Adds an existing attribute to an element.

Use this method to add an existing attribute to an element. oNewAttr is the attribute to be added. If the element already contains an attribute having the same name, that attribute is removed from the element. Note that the search for an existing attribute does not take namespaces into account. Use the SetAttributeNodeNS method if namespaces must be taken into account.

To declare a namespace prefix for an element, set the attribute's Name to the value "xmlns:prefix" where "prefix" is the prefix for the namespace. Set the attribute's Value property to the Unique Resource Identifier (URI) of the namespace. This method adds the namespace to the element's list of namespaces.

To declare the default namespace for the element, set the attribute's Name property to the value "xmlns" and the attribute's Value to the namespace URI.

This method returns the attribute that was added to the element.

See also: TXpAttribute.Name, SetAttribute, SetAttributeNodeNS, SetAttributeNS, TXpAttribute.Value

SetAttributeNodeNS method

```
function SetAttributeNodeNS(
  oNewAttr : TXpAttribute) : TXpAttribute;
```

Adds an existing attribute to an element, taking namespaces into account.

Use this method to add an existing attribute to an element. oNewAttr is the attribute to be added. If the element already contains an attribute having the same name, that attribute is removed from the element. The search for an existing attribute takes namespaces into account.

To declare a namespace prefix for an element, set the attribute's Name property to the value "xmlns:prefix" where "prefix" is the prefix for the namespace. Set the attribute's Value property to the Unique Resource Identifier (URI) of the namespace. This method adds the namespace to the element's list of namespaces.

To declare the default namespace for the element, set the attribute's Name to the value "xmlns" and the attribute's Value to the namespace URI.

This method returns the attribute that was added to the element.

See also: TXpAttribute.Name, SetAttribute, SetAttributeNode, SetAttributeNS, TXpAttribute.Value

SetAttributeNS method

```
procedure SetAttributeNS(
  sNamespaceURI, sLocalName, sValue : DOMString);
```

Creates a new attribute and adds it to the element.

Use this method to add a named attribute to an element. sNamespaceURI is the URI of the attribute's namespace. sLocalName is the attribute's name excluding a prefix. sValue is the value of the attribute. If the element already contains an attribute having the same name then this method removes that attribute before adding the new attribute. The search for an existing attribute takes namespaces into account.

To declare a namespace prefix for an element, set sName to the value "xmlns:prefix" where "prefix" is the prefix for the namespace. Set sValue to the Unique Resource Identifier (URI) of the namespace. This method adds the namespace to the element's list of namespaces.

To declare the default namespace for the element, set sName to the value "xmlns" and sValue to the namespace URI.

See also: SetAttribute, SetAttributeNode, SetAttributeNodeNS

TagName — property

```
property TagName : DOMString
```

Default: Empty string

Returns the name of the element.

This property is just another way to access the value of the element's NodeName.

See also: TXpNode.NodeName

TXpText Class

The TXpText class implements the Text interface as defined in the DOM specification. An instance of TXpText represents the textual content (i.e., character data) of an element or attribute and has an instance of TXpElement or TXpAttr as its parent node.

If there is no markup inside an element's content, the text is contained in a single instance of TXpText. If there is markup inside an element's content, the textual content is parsed into a list of TXpElement and TXpText instances. The instances are added as children of the element.

When the object model first loads a document, there is only one instance of class TXpText per block of text. An application may insert adjacent blocks of text or split an existing text instance using the SplitText method of this class. However, there is no way to represent this structure if the document is written to disk. Therefore, adjacent blocks of text are combined into one block of text in the written document. Optionally, the application may use the Normalize method of the TXpElement class to merge adjacent blocks of text.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 139
  - ❷ TXpCharacterData (XPDOM) 150
    - TXpText (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❷ Data
- ❶ FirstChild
- ❶ LastChild
- ❷ Length
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❷ AppendData
- ❶ CloneNode
- ❷ DeleteData
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes

❶ HasChildNodes
❶ InsertBefore
❷ InsertData
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
❷ ReplaceData
SplitText
❷ SubstringData
❶ Supports

4

SplitText — method

```
function SplitText(wOffset : Integer) : TXpText;
```

Breaks the text node into two text nodes at the specified offset.

Use this method to break a text node into two text nodes. wOffset is the zero-based position at which the text is to be split. This method creates a new text node, marks it as the sibling of the current text node, and moves the text starting at position wOffset into the new node. This function returns the new node.

Note that if the document is output to a file or memory, the adjacent text nodes are still written as one text node.

TXpComment Class

The TXpComment class implements the Comment interface as defined in the DOM specification. An instance of this class represents an XML comment which is all the text between the tags “<!--” and “-->”. A comment may be placed anywhere in an XML document.

This class does not introduce any new properties, methods, or events.

4

Hierarchy

TObject (VCL/CLX)

❶ TXpNode (XPDOM). 139

❷ TXpCharacterData (XPDOM) 150

TXpComment (XPDOM)

Properties

❶ Attributes
❶ ChildNodes
❷ Data
❶ FirstChild
❶ LastChild
❷ Length
❶ LevelCode
❶ LocalName
❶ NamespaceURI
❶ NextSibling
❶ NodeID
❶ NodeName
❶ NodeStringType
❶ NodeType
❶ NodeValue
❶ OwnerDocument
❶ ParentNode
❶ Prefix
❶ PreviousSibling
❶ Text
❶ XmlDocument

Methods

❶ AddRef
❶ AppendChild
❷ AppendData
❶ CloneNode
❷ DeleteData
❶ ForceOwnerDocument
❶ GetChildNodesByNodeType
❶ GetNodesByNodeType
❶ HasAttributes
❶ HasChildNodes
❶ InsertBefore
❷ InsertData
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
❷ ReplaceData
❷ SubstringData
❶ Supports

Chapter 5: Document Object Model Extended Interfaces

This chapter defines the XMLPartner classes that implement the extended interfaces of the Document Object Model (DOM) Level 1 Specification. There is a fairly close match between the XMLPartner class declarations and the extended interfaces in the DOM specification. XMLPartner also introduces a number of classes specifically geared to the evaluation and construction of Document Type Definitions (DTD). Note that these classes are a TurboPower Software extension and are not part of the official DOM specification.

Table 5.1 shows the correspondence of the DOM interfaces to the XMLPartner classes.

Table 5.1: *DOM interface with corresponding XMLPartner classes*

DOM Interface	XMLPartner Class
CDATASection	TXpCDATASection
DocumentType	TXpDocumentType
Notation	TXpDTDNotation
Entity	TXpDTDEntity
EntityReference	TXpEntityReference
ProcessingInstruction	TXpProcessingInstruction

Table 5.2 lists the DTD-specific classes introduced by XMLPartner

Table 5.2: *XMLPartner classes*

XMLPartner Class	Purpose
TXpDTDElement	Represents an element declaration within the DTD.
TXpDTDElementGroup	Represents a group of content specifications within an element declaration.
TXpDTDElementContent	Represents a content specification within an element declaration.
TXpDTDAttList	Represents an attribute list declaration within the DTD.
TXpDTDAttDefinition	Represents an attribute definition within an attribute list.
TXpDTDConditional	Represents a conditional declaration within the DTD.

Building a DTD dynamically

When creating an XML document from scratch, you can dynamically construct the Document Type Definition (DTD) using the DTD-related classes defined in this chapter. Note that what you build must follow the Document Type rules in the XML specification. For example, you may not add an instance of TXpDTDAttList as a child of an element declaration.

There are two limitations to the dynamic creation of a DTD:

1. Entities may be defined in the DTD. However, the XMLPartner DOM implementation resolves the entity references in the XML document.
2. Parameter entities are resolved in the DTDs and remain so when the document is written to a file.

The remainder of this section leads you through the dynamic creation of a DTD. The code assumes that your application has already instantiated an instance of TXpObjModel and that the object model contains an instance of TXpDocument.

The goal is to create an object structure that emulates the following DTD:

```
<!DOCTYPE doc [
  <!ELEMENT doc (contact+)>
  <!ELEMENT contact ANY>
  <!ATTLIST contact name CDATA #IMPLIED>
]>
```

The first step is to instantiate a document type having the name “doc”:

```
aDocType := aObjModel.Document.CreateDocumentType('doc');
```

Next, an instance of TXpDTDElement is created to represent the declaration of element “doc.” Note that the example appends the element to the document type object and calls the element’s Release method:

```
aDTDElem := aDocType.CreateDTDElement('doc', CONTENT_ELEMENTS);
aDocType.AppendChild(aDTDElem);
aDTDElem.Release;
```

The element declaration contains an element group consisting of the element “contact.” The following example instantiates the element group and adds it to the DTD tree as a child of the “doc” element declaration:

```
aDTDElemGroup := aDTDElem.CreateDTDElementGroup;
aDTDElem.AppendChild(aDTDElemGroup);
aDTDElemGroup.Release;
```

To complete the “doc” element declaration, the source code creates an element content instance representing the element “contact.” The element content instance is added to the DTD tree as a child of the element group:

```
aDTDElemItem := aDTDElemGroup.CreateDTDElementContent(
  'contact', OCCURS_REQ_REPEAT, REL_OR);
aDTDElemGroup.AppendChild(aDTDElemItem);
aDTDElemItem.Release;
```

The following example adds an element declaration for “contact”:

```
aDTDElem := aDocType.CreateDTDElement('contact', CONTENT_ANY);
aDocType.AppendChild(aDTDElem);
aDTDElem.Release;
```


The following example defines an attribute list for element “contact” and adds the attribute list to the document tree as a child of the document type:

```
aDTDAttlist := aDocType.CreateDTDAttlist('contact');
aDocType.AppendChild(aDTDAttlist);
aDTDAttlist.Release;
```

To complete the “contact” attribute list declaration, the following example creates an attribute definition and adds it to the attribute list:

```
aDTDAttDef := aDTDAttlist.CreateDTDAttDefinition(
  'name', ATTRIBUTE_CDATA, nil, ATTRIBUTE_DEFAULT_IMPLIED, '');
aDTDAttlist.AppendChild(aDTDAttDef);
aDTDAttDef.Release;
```

With the element and attribute declarations complete, the following example attaches the document type to the XML document and calls the document type’s Release method:

```
aObjModel.Document.AppendChild(aDocType);
aDocType.Release;
```

If the XML object’s SaveToFile method is called, the DTD will be written to the file in the format shown earlier in this example.

TXpCDATASection Class

The TXpCDATASection class implements the CDATASection interface defined in the DOM specification. Use CDATA sections to contain blocks of text that should not be evaluated by the parser. CDATA sections cannot be nested nor can they contain child nodes. Their primary purpose is to include material such as XML or HTML fragments without having to escape all the delimiters.

This class does not introduce any new properties, methods, or events.

Hierarchy

5

TObject (VCL/CLX)

❶ TXpNode (XPDOM) 135

❷ TXpCharacterData (XPDOM) 146

❸ TXpText (XPDOM) 171

TXpCDATASection (XPDOM)

Properties

❶ Attributes
❶ ChildNodes
❷ Data
❶ FirstChild
❶ LastChild
❷ Length
❶ LevelCode
❶ LocalName
❶ NamespaceURI
❶ NextSibling
❶ NodeID
❶ NodeName
❶ NodeStringType
❶ NodeType
❶ NodeValue
❶ OwnerDocument
❶ ParentNode
❶ Prefix
❶ PreviousSibling
❶ Text
❶ XmlDocument

Methods

❶ AddRef
❶ AppendChild
❷ AppendData
❶ CloneNode
❷ DeleteData
❶ ForceOwnerDocument
❶ GetChildNodesByNodeType
❶ GetNodesByNodeType
❶ HasAttributes
❶ HasChildNodes
❶ InsertBefore
❷ InsertData
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
❷ ReplaceData
❸ SplitText
❷ SubstringData
❶ Supports

TXpDocumentType Class

The TXpDocumentType class implements the DocumentType interface declared in the DOM specification. Class TXpDocumentType represents an external or internal Document Type Definition (DTD) describing the structure of the document. Each instance of TXpDocument may or may not have a DTD. If a document does have a DTD, the instance of TXpDocumentType representing the DTD is available through the DocType property of class TXpDocument.

The IsExternalDTD property indicates whether or not the DTD is external to the XML document. If the DTD is external then the current instance of TXpDocumentType does not have any child nodes. To access the external DTD and its child nodes, use the ExternalDTD property of this class.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDocumentType (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ExternalDTD
- ❶ FirstChild
- IsExternalDTD
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- CreateDTDAttList
- CreateDTDConditional
- CreateDTDElement
- CreateDTDEntity
- CreateDTDNotation
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

CreateDTDAttList — method

```
function CreateDTDAttlist(sName : DOMString) : TXpDTDAttlist;
```

Instantiates a DTD attribute list.

Use this method to create an attribute list for a DTD. sName is the name of the attribute list. This function returns an instance of TXpDTDAttList that does not contain any attributes. The attribute list must be attached to the DTD using the AppendChild or InsertBefore methods.

Use the CreateDTDAttDefinition method of the TXpDTDAttList class to create attributes for the attribute list.

See also: TXpNode.AppendChild, TXpDTDAttList.CreateDTDAttDefinition, TXpNode.InsertBefore

CreateDTDConditional — method

```
function CreateDTDConditional(wType : Integer) : TXpDTDConditional;
```

Creates a conditional section of the specified type.

Use this method to create a conditional section. wType indicates which kind of conditional section to create and may have the following values:

Constant	Value
CONDITIONAL_INCLUDE	0
CONDITIONAL_IGNORE	1

This function returns an instance of TXpDTDConditional. The conditional section node must be attached to the DTD using the AppendChild or InsertBefore methods of TXpNode.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

CreateDTDElement — method

```
function CreateDTDElement(
  sName : DOMString; wContentType : Integer) : TXpDTDElement;
```

Creates an ELEMENT declaration for a DTD.

Use this method to create an ELEMENT declaration. sName is the element's name and wContentType identifies the type of information in the element. wContentType may have the following values:

Constant	Value
CONTENT_ANY	1
CONTENT_EMPTY	2
CONTENT_MIXED	3
CONTENT_ELEMENTS	4

This function returns an instance of TXpDTDElement. The element node must be attached to the DTD using the AppendChild or InsertBefore methods of TXpNode.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

CreateDTDEntity — method

```
function CreateDTDEntity(sName : DOMString) : TXpDTDEntity;
```

Creates an ENTITY declaration.

Use this method to create an ENTITY declaration for the DTD. sName is name of the entity. This function returns an instance of TXpDTDEntity. The entity node must be attached to the DTD using the AppendChild or InsertBefore methods of TXpNode.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

CreateDTDNotation — method

```
function CreateDTDNotation(sName : DOMString) : TXpDTDNotation;
```

Creates a NOTATION declaration.

Use this method to create a NOTATION declaration for the DTD. sName is the name of the notation. This function returns an instance of TXpDTDNotation. The notation node must be attached to the DTD using the AppendChild or InsertBefore methods of TXpNode.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

ExternalDTD — read-only property

```
property ExternalDTD : TXpDocumentType
```

Provides access to an external DTD.

If the parsed document has an external DTD, use this property to access the DTD. To determine if the DTD is external, use the IsExternalDTD property.

See also: IsExternalDTD

IsExternalDTD — read-only property


```
property IsExternalDTD : Boolean
```

Indicates whether the DTD is stored in a file external to the parsed document.

Use this property to determine if the DTD is external or internal. This property returns True if the DTD is external to the parsed document. Otherwise, it returns False.

See also: ExternalDTD

Name — property

```
property Name : DOMString
```

Returns the name of the document type.

TXpDTDElement Class

This class represents an ELEMENT declaration within the Document Type Definition (DTD). Do not directly create instances of this class. Instead, use the CreateDTDElement method of the TXpDocumentType class.

The Name property defines the name of the element. The ContentType property defines the content specification in the element declaration. If the content specification references one or more child elements or occurrences of PCDATA, the ChildNodes property will contain an instance of the TXpDTDElementGroup class.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135

  TXpDTDElement (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ContentType
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- CreateDTDElementGroup
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

ContentType — property

```
property ContentType : Integer
```

Default: CONTENT_ANY

Identifies the content type of the element declaration.

Use this property to determine the content type specified in the element declaration. The following table lists the legal values for this property:

Constant	Value	Meaning
CONTENT_ANY	1	The element declaration contains any mix of character data and child elements.
CONTENT_EMPTY	2	The element declaration does not contain any character data or child elements.
CONTENT_MIXED	3	The element declaration contains character data or a list of elements repeated in any order with any amount of PCDATA mixed in between the elements.
CONTENT_ELEMENTS	4	The element declaration contains only child elements.

CreateDTDElementGroup — method

```
function CreateDTDElementGroup : TXpDTDElementGroup;
```

Creates an element group for use with an element declaration.

Use this method to create an element group for use inside an element declaration. This method returns an empty element group. After creating the element group, attach it to an element declaration using the AppendChild or InsertBefore method of the TXpNode class.

See also: TXpNode.AppendChild, ContentType, TXpNode.InsertBefore

Name — property

```
property Name : DOMString
```

Provides access to the name of the element in the element declaration.

Use this property to get or set the name of the element.

TXpDTDElementGroup Class

The TXpDTDElementGroup class represents a group of PCDATA and elements within a content specification of an element declaration. An instance of TXpDTDElementGroup will be a child of an instance of TXpDTDElement. The ChildNodes property of an instance of TXpDTDElementGroup will contain one or more instances of TXpDTDElementContent. If the content specification includes nested groups of elements and PCDATA, the ChildNodes property may also contain one or more instances of class TXpDTDElementGroup.

Do not create instances of TXpDTDElementGroup directly. Instead, use the CreateDTDElementGroup method of the TXpDTDElement class. To create a nested element group, use the CreateDTDElementGroup method of this class.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDElementGroup (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- Occurs
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- CreateDTDElementContent
- CreateDTDElementGroup
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

CreateDTDElementContent method

```
function CreateDTDElementContent(sName : DOMString;
  wOccurs, wRelation : Integer) : TXpDTDElementContent;
```

Creates an element content instance.

Use this method to create an element content instance for use in an element content group. sName is the name of the element type that may exist within the element content group. wOccurs indicates if the element type is required and repeatable. The following table lists the legal values for wOccurs:

Constant	Value	Meaning
OCCURS_REQ_NOREPEAT	0	The element must occur within the parent element and it may not be repeated.
OCCURS_OPT_NOREPEAT	1	The element is optional. If it does occur within the parent element then it may not be repeated.
OCCURS_OPT_REPEAT	2	The element is optional. If it does occur within the parent element then it may be repeated.
OCCURS_REQ_REPEAT	3	The element is required to occur within the parent element. The element may be repeated.

wRelation indicates how the element type relates to a subsequent element type. The following table lists the legal values for wRelation:

Constant	Value	Meaning
REL_OR	0	The parent element may contain this element or the following element.
REL_AND	1	The parent element contains this element and the following element.

This method returns an instance of TXpDTDElementContent. Add the instance to an element group using the AppendChild or InsertBefore methods of the TXpNode class.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

CreateDTDElementGroup — method

```
function CreateDTDElementGroup : TXpDTDElementGroup;
```

Creates a DTD element group.

Use this method to create nested DTD element groups. This method returns an empty element group that must be attached to an existing element group using the AppendChild or InsertBefore methods of the TXpNode class.

See also: TXpNode.AppendChild, TXpNode.InsertBefore

Occurs — property

```
property Occurs : Integer
```

Default: OCCURS_REQ_NONREPEAT

Specifies how the element group may occur within the document.

Use this property to determine whether the element group is optional, required, repeatable, or non-repeatable. The following table lists the legal values for this property:

Constant	Value	Meaning
OCCURS_REQ_NOREPEAT	0	The element must occur within the parent element and it may not be repeated.
OCCURS_OPT_NOREPEAT	1	The element is optional. If it does occur within the parent element then it may not be repeated.
OCCURS_OPT_REPEAT	2	The element is optional. If it does occur within the parent element then it may be repeated.
OCCURS_REQ_REPEAT	3	The element is required to occur within the parent element. The element may be repeated.

TXpDTDElementContent Class

The TXpDTDElementContent class represents an element or PCDATA declaration within the content portion of an element declaration. When the ContentType property of an instance of the TXpDTDElement class is set to CONTENT_MIXED or CONTENT_ELEMENTS, the TXpDTDElement has a child node of the TXpDTDElementGroup class. Within TXpDTDElementGroup, there will be one or more instances of TXpDTDElementContent.

Do not create instances of TXpDTDElementContent directly. Instead, use the CreateDTDElementContent method of the TXpDTDElementGroup class.

Each instance of TXpDTDElementContent identifies whether the content is PCDATA or a named element, whether the content is required or repeatable, and how the content is related to its NextSibling content instance.

For example, assume DTD contains the following element declaration:

```
<!ELEMENT list (#PCDATA|item)*>
```

Figure 5.1 shows the object structure after the element declaration is parsed.

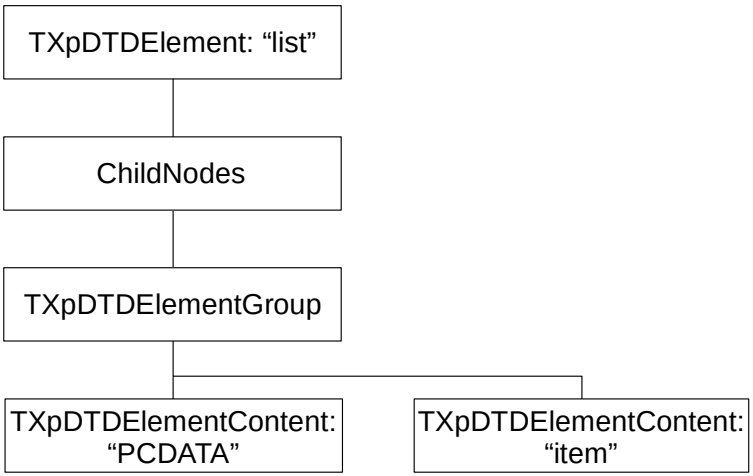


Figure 5.1: Object structure after data element declaration is parsed

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDElementContent (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- Occurs
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- Relation
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

Reference Section

Name — property

```
property Name : DOMString
```

Default: Empty string

Provides access to the name of an element in an element content group.

Depending upon its ContentType, an element declaration may refer to one or more instances of PCDATA and one or more element types. If the element content is PCDATA then this property returns the value “#PCDATA”. Otherwise, it returns the name of the referenced element type.

See also: TXpDTDElement.ContentType

Occurs — property

```
property Occurs : Integer
```

Default: OCCURS_REQ_NOREPEAT

Specifies how the element may occur within the document.

Use this property to determine is optional, required, repeatable, or non-repeatable. The following table lists the legal values for this property:

Constant	Value	Meaning
OCCURS_REQ_NOREPEAT	0	The element must occur within the parent element and it may not be repeated.
OCCURS_OPT_NOREPEAT	1	The element is optional. If it does occur within the parent element then it may not be repeated.
OCCURS_OPT_REPEAT	2	The element is optional. If it does occur within the parent element then it may be repeated.
OCCURS_REQ_REPEAT	3	The element is required to occur within the parent element. The element may be repeated.

```
property Relation : Integer
```

Default: REL_OR

Specifies how the element relates to the following.

The following table lists the legal values for this property:

Constant	Value	Meaning
REL_OR	0	The parent element may contain this element or the following element.
REL_AND	1	The parent element contains this element and the following element.

TXpDTDAttList Class

The TXpDTDAttList class represents an attribute list declaration within a Document Type Definition (DTD). An instance of the TXpDTDAttList class has an instance of TXpDocumentType as its parent node and is associated with a specific element declaration via the Name property. An instance of this class will have one or more attribute definitions. The attribute definitions are implemented using instances of TXpDTDAttDefinition and those instances are accessible via this class's ChildNodes property.

Hierarchy

5

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDAttList (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- CreateDTDAttDefinition
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

CreateDTDAttDefinition method

```
function CreateDTDAttDefinition(
  sName : DOMString; wAttrType : Integer;
  oEnumeration : TStringList; wValueType : Integer;
  sValue : DOMString) : TXpDTDAttDefinition;
```

Creates an attribute definition that occurs within an attribute list declaration.

Use this method to create a new attribute definition. sName is the name of the attribute. wAttrType is the attribute's type. The valid attribute types are shown in the following table:

Constant	Value	Meaning
ATTRIBUTE_UNDECLARED	0	The attribute type is undeclared. The attribute declaration has this value when it is first created.
ATTRIBUTE_CDATA	1	The attribute may be assigned a string.
ATTRIBUTE_ID	2	The attribute is an ID attribute, meaning that the attribute's value uniquely identifies the element with which it is associated. No other element may have the same value for its ID attribute.
ATTRIBUTE_IDREF	3	The attribute is a reference to another element's ID attribute.
ATTRIBUTE_IDREFS	4	Identical to ATTRIBUTE_IDREF but the attribute may include references to multiple elements.
ATTRIBUTE_ENTITY	5	The attribute must contain the name of a properly declared unparsed entity.
ATTRIBUTE_ENTITIES	6	Identical to ATTRIBUTE_ENTITY but the attribute may include the names of multiple entities.
ATTRIBUTE_NMTOKEN	7	The attribute may be assigned any sequence of characters that match the Nmtoken production.
ATTRIBUTE_NMTOKENS	8	Identical to ATTRIBUTE_NMTOKEN but the attribute may be assigned multiple Nmtoken values.

Constant	Value	Meaning
ATTRIBUTE_ENUMERATED	9	One or more enumerated values have been defined in the attribute declaration and the attribute must be assigned one of those values.
ATTRIBUTE_NOTATION	10	The attribute must be assigned one of the notation values specified in the attribute declaration.

oEnumeration is required only when wAttrType is set to ATTRIBUTE_ENUMERATED; otherwise it may be set to nil. When specified, the instance of TStringList referenced by oEnumeration must contain the valid enumerated values for the attribute.

wValueType indicates whether the default value is specified or required. The following table lists the legal values for wValueType:

Constant	Value	Meaning
ATTRIBUTE_DEFAULT_UNDECLARED	0	A value for this property has not been specified. This situation occurs when the attribute declaration is first created.
ATTRIBUTE_DEFAULT_SPECIFIED	1	A default value was specified for the attribute.
ATTRIBUTE_DEFAULT_IMPLIED	2	The attribute need not be specified. If it isn't then the application implies a value.
ATTRIBUTE_DEFAULT_REQUIRED	3	For every element of this type, a value must be specified for the attribute.
ATTRIBUTE_DEFAULT_FIXED	4	The default value given for the attribute is the only value allowed for the attribute.

If the wValueType has the value ATTRIBUTE_DEFAULT_SPECIFIED or ATTRIBUTE_DEFAULT_FIXED, then sValue specifies the default value for the attribute. Otherwise, the value of sValue is ignored and may be set to an empty string.

Name property

```
property Name : DOMString
```

Default: Empty string

Returns the name of the element with which the attribute list is associated.

The attribute list declaration references a specific element declaration. Use this property to retrieve or set the name of that element declaration.

TXpDTDAttDefinition Class

The TXpDTDAttDefinition class represents one attribute definition that is a part of an attribute list declaration. Do not dynamically create instances of TXpDTDAttDefinition. Instead, use the CreateDTDAttDefinition method of TxpDTDAttList. Each instance of TXpDTDAttDefinition will have an instance of TXpDTDAttList as its parent node.

The name of the attribute is available through the Name property. The default type and default value are available through the DefaultType and DefaultValue properties, respectively. The AttType property defines the type of data that may be assigned to an instance of the attribute. If the attribute is an enumerated type, the enumerated values are available through the Enumeration property.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDAttDefinition (XPDOM)

Properties

- ❶ Attributes
- AttType
- ❶ ChildNodes
- DefaultType
- DefaultValue
- Enumeration
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

AttType — property

```
property AttType : Integer
```

Default: ATTRIBUTE_UNDECLARED

Identifies the attribute type specified in the attribute declaration.

Use this property to identify and control the type of data that may be specified for an attribute. The following table lists the legal values for this property:

Constant	Value	Meaning
ATTRIBUTE_UNDECLARED	0	The attribute type is undeclared. The attribute declaration has this value when it is first created.
ATTRIBUTE_CDATA	1	The attribute may be assigned a string.
ATTRIBUTE_ID	2	The attribute is an ID attribute, meaning that the attribute's value uniquely identifies the element with which it is associated. No other element may have the same value for its ID attribute.
ATTRIBUTE_IDREF	3	The attribute is a reference to another element's ID attribute.
ATTRIBUTE_IDREFS	4	Identical to ATTRIBUTE_IDREF but the attribute may include references to multiple elements.
ATTRIBUTE_ENTITY	5	The attribute must contain the name of a properly declared unparsed entity.
ATTRIBUTE_ENTITIES	6	Identical to ATTRIBUTE_ENTITY but the attribute may include the names of multiple entities.
ATTRIBUTE_NMTOKEN	7	The attribute may be assigned any sequence of characters that match the Nmtoken production.
ATTRIBUTE_NMTOKENS	8	Identical to ATTRIBUTE_NMTOKEN but the attribute may be assigned multiple Nmtoken values.

Constant	Value	Meaning
ATTRIBUTE_ENUMERATED	9	One or more enumerated values have been defined in the attribute declaration and the attribute must be assigned one of those values.
ATTRIBUTE_NOTATION	10	The attribute must be assigned one of the notation values specified in the attribute declaration.

DefaultType property

5

```
property DefaultType : Integer
```

Default: ATTRIBUTE_DEFAULT_UNDECLARED

Indicates whether the attribute's presence is required.

Use this property to determine if the attribute is required and how the XML processor should react if it is not present in the document. The following table lists the legal values for this property:

Constant	Value	Meaning
ATTRIBUTE_DEFAULT_UNDECLARED	0	A value for this property has not been specified. This situation occurs when the attribute declaration is first created.
ATTRIBUTE_DEFAULT_SPECIFIED	1	A default value was specified for the attribute.
ATTRIBUTE_DEFAULT_IMPLIED	2	The attribute need not be specified. If it isn't then the application implies a value.
ATTRIBUTE_DEFAULT_REQUIRED	3	For every element of this type, a value must be specified for the attribute.
ATTRIBUTE_DEFAULT_FIXED	4	The default value given for the attribute is the only value allowed for the attribute.

See also: DefaultValue

DefaultValue — property

```
property DefaultValue : DOMString
```

Default: Empty string

Identifies the default value found in the attribute declaration.

If the DefaultType property is set to ATTRIBUTE_DEFAULT_SPECIFIED or ATTRIBUTE_DEFAULT_FIXED then this property returns the default value found in the attribute declaration. Otherwise, this property returns an empty string.

See also: DefaultType

Enumeration — read-only property

```
property Enumeration : TStringList
```

Contains the enumeration values found in the attribute declaration.

This property applies only to attribute declarations having their AttType property set to ATTRIBUTE_ENUMERATED. This property returns an instance of TStringList containing the enumeration values found in the attribute declaration. This property is read-only in the sense that the instance of TStringList may not be changed by the application using XMLPartner. However, the application may change the values within the instance of TStringList.

See also: AttType

Name — property

```
property Name : DOMString
```

Default: Empty string

Returns the name of the attribute in the attribute declaration.

Use this property to get or set the name of the attribute in the attribute declaration.

TXpDTDEntity Class

The TXpDTDEntity class represents an entity declaration within a Document Type Definition (DTD). Each instance of this class has an instance of TXpDocumentType as its parent node. Do not dynamically create instances of this class. Instead, use the CreateDTDEntity method of the TXpDocumentType class.

An entity declaration may define an internal or external entity. Optionally, the entity may be a general entity or a parameter entity. The IsPE property of a parameter entity returns True. The IsPE property of a general entity returns False.

5

An internal entity's name and value are available via the Name and Value properties, respectively. An external entity may have a PublicID and/or a SystemID. If the external entity contains a notation tag, the NotationName property contains the name of the notation.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDEntity (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- IsPE
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- NotationName
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- PublicID
- SystemID
- ❶ Text
- Value
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll

- RemoveChild
- ReplaceChild
- Supports

5

IsPE — property

```
property IsPE : Boolean
```

Default: False

Indicates whether the entity is a parameter entity.

If the entity is declared as a parameter entity, this property returns True. Otherwise, this property returns False.

Name — property

```
property Name : DOMString
```

Default: Empty string

Defines the name of the entity.

Use this property to get or set the name of the entity.

NotationName — property

```
property NotationName : DOMString
```

Default: Empty string

Defines the name of the notation referenced in the entity declaration.

For internal entities, this property returns an empty string. For external entities with an NDATA tag, this property returns the name of the referenced notation.

PublicID — property

```
property PublicID : DOMString
```

Default: Empty string

Identifies the public ID of the notation.

Use this property to get or set the notation's public ID. For internal entities, this property returns an empty string.

SystemID — property

```
property SystemID : DOMString
```

Default: Empty string

Identifies the system ID of the notation.

Use this property to get or set the notation's system ID. For internal entities, this property returns an empty string.

Value — property

```
property Value : DOMString
```

Default: Empty string

Identifies the value assigned to the entity.

Use this property to get or set the entity's value. If the entity is external, this property returns an empty string.

TXpDTDNotation Class

The TXpDTDNotation class represents a Notation declaration within a Document Type Definition (DTD). An instance of this class will have an instance of TXpDocumentType as its ParentNode. Do not dynamically create instances of this class. Instead, use the CreateDTDNotation method of the TXpDocumentType class.

Instances of this class may be referenced by instances of the TXpDTDEntity class via the NotationName property of TXpDTDEntity.

Hierarchy

5

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpDTDNotation (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- Name
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- PublicID
- SystemID
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

Reference Section

Name — property

```
property Name : DOMString
```

Default: Empty string

Defines the name of the notation.

Use this property to get or set the notation's name. For example, the name of the notation shown in the following line is "png":

```
<!NOTATION png SYSTEM "http://www.someStuff.com/pngViewer.exe">
```

PublicID — property

```
property PublicID : DOMString
```

Default: Empty string

Identifies the public ID of the notation.

Use this property to get or set the notation's public ID.

SystemID — property

```
property SystemID : DOMString
```

Default: Empty string

Identifies the system ID of the notation.

Use this property to get or set the notation's system ID.

TXpDTDConditional Class

The TXpDTDConditional class represents a conditional declaration within the Document Type Definition (DTD). An instance of this class will have an instance of TXpDocumentType as its ParentNode. Do not dynamically create instances of this class. Instead, use the CreateDTDConditional method of the TXpDocumentType class.

Hierarchy

TObject (VCL/CLX)

❶ TXpNode (XPDOM). 135

❷ TXpDocumentType (XPDOM) 183

TXpDTDConditional (XPDOM)

Properties

❶ Attributes
❶ ChildNodes
CondType
❷ ExternalDTD
❶ FirstChild
❷ IsExternalDTD
❶ LastChild
❶ LevelCode
❶ LocalName
❷ Name
❶ NamespaceURI
❶ NextSibling
❶ NodeID
❶ NodeName
❶ NodeStringType
❶ NodeType
❶ NodeValue
❶ OwnerDocument
❶ ParentNode
❶ Prefix
❶ PreviousSibling
❶ Text
❶ XmlDocument

Methods

❶ AddRef
❶ AppendChild
❶ CloneNode
❷ CreateDTDAttList
❷ CreateDTDConditional
❷ CreateDTDElement
❷ CreateDTDEntity
❷ CreateDTDNotation
❶ ForceOwnerDocument
❶ GetChildNodesByNodeType
❶ GetNodesByNodeType
❶ HasAttributes
❶ HasChildNodes
❶ InsertBefore
❶ IsAfter
❶ Release
❶ RemoveAll
❶ RemoveChild
❶ ReplaceChild
❶ Supports

CondType — property

```
property CondType : Integer
```

Default: CONDITIONAL_INCLUDE

Specifies the type of conditional section.

Use this property to get or set the type in a conditional section declaration. The following table lists the valid values for this property:

Constant	Value	Meaning
CONDITIONAL_INCLUDE	0	Declarations within the include section are to be used for validation.
CONDITIONAL_IGNORE	1	Declarations within the ignore section are read but ignored by the parser.

TXpEntityReference Class

The TXpEntityReference class implements the EntityReference interface defined in the DOM specification. An instance of this class refers to an instance of the TXpDTDEntity class. At the time of this writing, the XMLPartner parser chooses to expand all entity references of loaded documents. In general, instances of this class will be encountered only when they are inserted by the application or when the document contains an unparsed entity reference.

This class does not implement any additional properties, methods, or events.

5

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpEntityReference (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

TXpProcessingInstruction Class

The TXpProcessingInstruction class implements the ProcessingInstruction interface as defined in the DOM specification. A *processing instruction* is used to keep application-specific information in the document. The processing instruction affects how the XML document is processed. Typically, the Target property of this class defines the application invoked by the processing instruction. The Data property typically defines the information passed to the Target application.

Hierarchy

TObject (VCL/CLX)

- ❶ TXpNode (XPDOM). 135
  - TXpProcessingInstruction (XPDOM)

Properties

- ❶ Attributes
- ❶ ChildNodes
- Data
- ❶ FirstChild
- ❶ LastChild
- ❶ LevelCode
- ❶ LocalName
- ❶ NamespaceURI
- ❶ NextSibling
- ❶ NodeID
- ❶ NodeName
- ❶ NodeStringType
- ❶ NodeType
- ❶ NodeValue
- ❶ OwnerDocument
- ❶ ParentNode
- ❶ Prefix
- ❶ PreviousSibling
- Target
- ❶ Text
- ❶ XmlDocument

Methods

- ❶ AddRef
- ❶ AppendChild
- ❶ CloneNode
- ❶ ForceOwnerDocument
- ❶ GetChildNodesByNodeType
- ❶ GetNodesByNodeType
- ❶ HasAttributes
- ❶ HasChildNodes
- ❶ InsertBefore
- ❶ IsAfter
- ❶ Release
- ❶ RemoveAll
- ❶ RemoveChild
- ❶ ReplaceChild
- ❶ Supports

Data — property

```
property Data : DOMString
```

Provides access to the content of the processing instruction.

The content of the processing instruction is defined as the first non white space character after the target to the character immediately preceding the '?>' tag. The Data property is typically used to contain the data passed to the application specified in the Target instruction.

Target — property

```
property Target : DOMString
```

Provides access to the target of the processing instruction.

The target is defined as the first token following the markup that begins the processing instruction. It is typically used to name an external application.

Chapter 6: An Overview of XSL

XML defines a language for organizing information. It focuses on structure and content instead of presentation. However, the information in an XML document doesn't have much value if it just sits there. The information is valuable when you use it to accomplish some purpose.

For example, a company's daily sales saved to an XML document could be sent to a warehouse application, causing product to be shipped to customers. The same XML document could also be used to build a weekly sales report or a commission report for the sales staff. The reports could be printed to paper, displayed on a computer monitor, viewed on a Web-enabled cell phone, or heard through a telephony application for order confirmations.

As you can see, a set of information may be used for multiple purposes. Each purpose has one or more output mediums and many ways in which the information could be presented through that medium. However, an XML document does not describe how its contents are to be presented to an audience. It says nothing of how it is to be formatted or which mediums it supports.

Stylesheets bridge the gap between content and presentation. Stylesheets provide a path to turn data into something of value, describing how information is to be formatted and displayed. The Extensible Stylesheet Language (XSL) is a language for describing stylesheets. As stated in the XSL specification, designers use an XSL stylesheet to express "how the source content should be styled, laid out, and paginated onto some presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book (W3C, 1999)."

The process of using XSL to convert an XML document into formatted output is comprised of two stages. The first stage, called *transformation*, constructs a *result tree* from an XML document. The result tree is a tree-like data structure, similar to the XML Document Object Model (DOM), which reflects the desired output. The second stage, called *formatting*, interprets the result tree into a format suitable for presentation in such mediums as a Web browser or printed page.

When the XSL specification was first created, it described both transformation and formatting. However, the specification grew so big that it was broken into three parts: XSL Transformations (XSLT), XPath, and XSL Formatting Objects (XSL FO). XSLT is a language for transforming XML documents into other XML documents (i.e., the result tree). XPath is a language for selecting data within an XML document. XSL FO is a language for describing the formatting of a result tree and is described in the XSL specification. The specification for each part is listed in Chapter 1.

XMLPartner Professional contains an XSL stylesheet processor and several filter components that are fully described in Chapter 7. The stylesheet processor uses an XML document and an XSL stylesheet to format and output the content of the XML document. The processor also supports embedded stylesheets. The filter components format the output of the result tree to HTML, RTF, Print Preview, or another XML document. Through its components, XMLPartner Professional supports XSLT Version 1.0 and a subset of the formatting object functionality described in XSL version 1.0. XMLPartner Professional also contains a query engine that supports XPath 1.0. For more details about the specific XSLT, XPath, and XSL FO instructions supported by XMLPartner Professional, see Chapters 8 and 9.

The remainder of this chapter contains examples designed to teach you how to create XSL stylesheets and how to process them using the components provided with XMLPartner Professional. Please note that the information in this chapter and the subsequent chapters pertain only to XMLPartner Professional and not to the XMLPartner product.

Creating a Stylesheet

A stylesheet is essentially an XML document with a grammar that looks like XML. A stylesheet may be embedded within an XML document, however the next few sections focus on external stylesheets. An external stylesheet is typically stored as a file with the extension "xsl". It contains an optional XML prolog and an xsl:stylesheet element that identifies it as a stylesheet. The following example shows an empty stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

Note that xsl:stylesheet is an element start tag and it has a corresponding end tag. The xmlns:xsl indicates that the stylesheet uses the xsl prefix for the namespace identified by the URI http://www.w3.org/1999/XSL/Transform. All XSL stylesheets must use this namespace URI. You may encounter stylesheets supported by Internet Explorer whose namespace URI is http://www.w3.org/TR/WD-xsl, which is the working draft of XSL. However, that URI is obsolete and should no longer be used.

The examples in this chapter create stylesheets for a fictitious company named Stuffmart. Stuffmart sells… stuff. It sells the stuff that you need as well as stuff you never dreamed of. The output produced by the examples is rather plain but it serves its purpose. The first stylesheet needed by Stuffmart is one that formats the employee of the month for each store location within the state of Colorado, United States. The resulting document is to be displayed as HTML on the company Web site. The Human Resources department creates the XML document containing the employees of the month. A copy of this month's document is installed in the Examples\Xmldocs of your XMLPartner Professional installation, under the name MonthEmp.xml. Following are the first few lines of this document:

```
<?xml version="1.0"?>
<EmployeesOfTheMonth>
  <Location name="Colorado Springs">
    <Employee name="Emily Martinez"
              title="Store manager"/>
  </Location>
  ...
</EmployeesOfTheMonth>
```

The document contains one Location element per Stuffmart location in the state of Colorado. Each Location element contains an Employee element. The Employee element uses name and title attributes to hold information about that location's employee of the month.

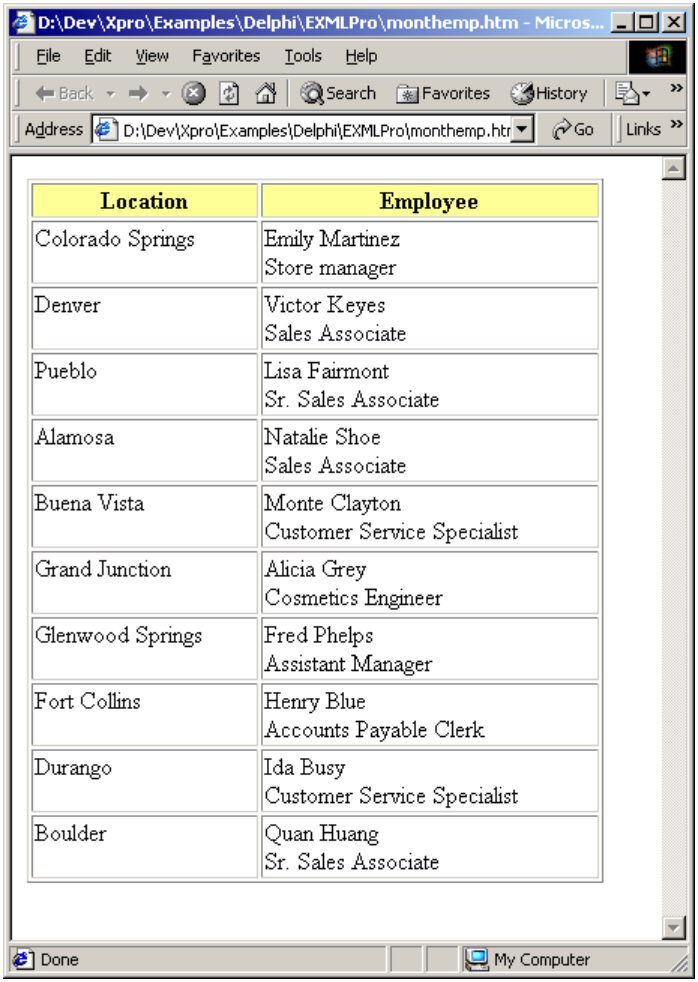
The Human Resources department wishes for the employees to be listed in tabular format within an HTML page. The first column of the table must identify the Stuffmart location. The second column must display the employee's name above their title. A stylesheet that meets these requirements is as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <table border="1" width="384">
    <tr bgcolor="#FFFF99">
      <th>Location</th>
      <th>Employee</th>
    </tr>
    <xsl:for-each select="/EmployeesOfTheMonth/Location">
      <tr valign="top">
        <td><xsl:value-of select="@name"/></td>
        <td>
          <xsl:value-of select="Employee/@name"/>
          <br/>
          <xsl:value-of select="Employee/@title"/>
      </td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>

</xsl:stylesheet>
```

This stylesheet is stored as MonthEmp.xsl in the same subfolder as MonthEmp.xml. To try out this example, start the EXMLPro editor in the Examples\Delphi\EXMLPro subfolder of your XMLPartner Professional installation. After the editor starts, open the MonthEmp.xml document. Once the document loads and displays, click on its XSL Preview tab. Verify that the HTML filter is selected and use the ellipsis button to select the MonthEmp.xsl stylesheet. Click the Render button. The XSL processor transforms the document into an HTML file and displays it in your default browser, as shown in Figure 6.1.



Location	Employee
Colorado Springs	Emily Martinez Store manager
Denver	Victor Keyes Sales Associate
Pueblo	Lisa Fairmont Sr. Sales Associate
Alamosa	Natalie Shoe Sales Associate
Buena Vista	Monte Clayton Customer Service Specialist
Grand Junction	Alicia Grey Cosmetics Engineer
Glenwood Springs	Fred Phelps Assistant Manager
Fort Collins	Henry Blue Accounts Payable Clerk
Durango	Ida Busy Customer Service Specialist
Boulder	Quan Huang Sr. Sales Associate

Figure 6.1: Employee of the month browser example

Each stylesheet contains one or more template rules that control how the XML document is transformed. A template rule consists of a pattern describing the nodes in the source document with which it is associated and a template that can be instantiated to form part of the result tree. Following is the template used in the Employee of the Month stylesheet:

```
<xsl:template match="/">
...
</xsl:template>
```

The match attribute of xsl:template uses an XPath expression to identify the elements or attributes that it transforms. In this example, the template matches the document root of the XML document. As defined in the XPath specification, the document root is the parent of the document element. In this situation, the document element is EmployeesOfTheMonth. Because this template matches the document root, it is executed when processing of the document first begins.

When the template executes, its contents tell the processor how to transform the document. The processor parses the contents of the template as though it were an XML document. Because the Human Resources department wants the final output to be displayed on a Web site, the body of this template contains HTML markup as shown in the following code snippet:

```
<table border="1" width="384">
    <tr bgcolor="#FFFF99">
      <th>Location</th>
      <th>Employee</th>
    </tr>
...
</table>
```

The XSL processor copies the HTML markup as-is to the result tree. The markup could just as well be XML element start and end tags. The XSL processor also copies the text "Location" and "Employee" to the result tree. As shown in the following code snippet, the template also contains several XSLT elements:

```
<xsl:for-each select="/EmployeesOfTheMonth/Location">
  <tr valign="top">
    <td><xsl:value-of select="@name"/></td>
    <td>
      <xsl:value-of select="Employee/@name"/>
      <br/>
      <xsl:value-of select="Employee/@title"/>
  </td>
  </tr>
</xsl:for-each>
```

Note that all XSLT elements must be well formed, having a start tag and an end tag or an empty-element tag. The first element encountered in the template is xsl:for-each. This element selects a set of nodes using the XPath expression given in its select attribute. The xsl:for-each element then executes its contents for each node matching the select attribute. In this example, xsl:for-each selects each Location element within the document. For each location, the template creates a new table row and fills it with information from the employee corresponding to that location.

The template obtains information about an employee using the xsl:value-of element. The select attribute of the xsl:value-of element uses an XPath expression to obtain a string value. The first instance of xsl:value-of selects the name attribute of the Location element being processed. The second and third instances select the name and title attributes of the Employee element within the Location element.

Using multiple templates

In the previous example, only one template was used to process the stylesheet. The output format closely matched the input data and the template selected information as necessary from the XML document to fill in the blanks. This is just one approach to creating a stylesheet. In this section, you will explore a new method for creating stylesheets.

The president of Stuffmart distributes an internal memo each month describing the state of the business. The memo is saved as an XML document, is posted to the internal Web site, and distributed in paper form to the location managers. The following example shows a typical press release. Some lines of the content element have been omitted to reduce the size of the example.

```
<?xml version="1.0"?>
<Memo distribution="internal">
  <from>Henry Stuff III</from>
  <to>All employees</to>
  <date>6 April 2001</date>
  <content>
  It is with great delight that I announce some results of our
  quarterly cost cutting efforts. Over this past quarter, the
  customer service segment of our business has managed to
  reduce customer returns by over <ital>50%</ital>. This has
  netted our business <emph>incredible</emph> savings and shows
  ...
  </content>
</Memo>
```

A copy of this memo can be found in the Examples\Xmldocs folder of your XMLPartner Professional installation in the file named StufMemo.xml. Notice that since the memo may be output to a Web site or paper, it does not contain any HTML-specific markup. Instead, it

segregates the information into from, to, date, and content elements. The content element uses an element named ital to indicate italicized text and emph to indicate bold text. The stylesheet used to process the memo can be found in the file BaseMemo.xsl in the Examples\Xmldocs subfolder. The stylesheet is as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <fo:block font-size="16pt" space-after="10">MEMO</fo:block>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="to">
    <fo:block font-size="10pt">
      <fo:inline font-weight="bold">
        <xsl:text>To:   </xsl:text>
      </fo:inline>
      <xsl:value-of expr="."/>
    </fo:block>
  </xsl:template>

  <xsl:template match="from">
    <fo:inline font-weight="bold">
      <xsl:text>From: </xsl:text>
    </fo:inline>
    <xsl:value-of expr="."/>
  </xsl:template>

  <xsl:template match="date">
    <fo:inline font-weight="bold">
      <xsl:text>Date: </xsl:text>
    </fo:inline>
    <xsl:value-of expr="."/>
  </xsl:template>

  <xsl:template match="content">
    <xsl:apply-templates/>
  </xsl:template>
```

```
  <xsl:template match="emph">
    <xsl:text/>
    <fo:inline font-weight="bold">
      <xsl:apply-templates/>
    </fo:inline>
  </xsl:template>

  <xsl:template match="ital">
    <xsl:text/>
    <fo:inline font-style="italic">
      <xsl:apply-templates/>
    </fo:inline>
  </xsl:template>

</xsl:stylesheet>
```

The stylesheet contains a template for the document root as well as for each element that may be found within the memo. The document root template is as follows:

```
<xsl:template match="/">
  <fo:block font-size="16pt" space-after="10">MEMO</fo:block>
  <xsl:apply-templates/>
</xsl:template>
```

Because this document may be output to HTML or the printed page, it uses the fo:block instruction to control font size. The font-size attribute of fo:block sets the font size to 16 points. The xsl:apply-templates element tells the processor to process the child nodes of the document root, applying the appropriate template to each node. The only child of the document root is the Memo element. The stylesheet does not contain a template for the Memo element. Fortunately, XSLT defines a built-in rule for element nodes. The following shows the built-in rule for element nodes:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

The built-in template matches all elements and the document root. Notice that the match expression uses the '|' character to indicate a logical or. The body of the template tells the processor to apply rules to the element's children. In this example, the processor looks for rules associated with the elements from, to, date, and content. The template matching the to element is shown in the following code snippet:

```
<xsl:template match="to">
  <fo:block font-size="10pt">
    <fo:inline font-weight="bold">
      <xsl:text>To:   </xsl:text>
    </fo:inline>
    <xsl:value-of expr="."/>
  </fo:block>
</xsl:template>
```


The template uses the font weight of the fo:inline element to change the weight of the current font to bold. After changing the weight of the font, the character string "To:" is output as is using the xsl:text element. The string value of the to element is then stored in the result tree using the xsl:value-of instruction. The expression '.' used by xsl:value-of tells the processor to use the current node. When a template is executed, it is executed in the context of a current node or nodes.

As the other elements are encountered, the appropriate template is found and executed by the XSL processor. In the first example of this chapter, the stylesheet sought after specific nodes. This could be called pull processing, where the stylesheet is pulling specific nodes into the result tree. In this example, the stylesheet is used as nodes are encountered within the document. This could be called push processing where the processor is asking the stylesheet if any of its templates are interested in the nodes.

Note that none of the templates affect any of the other templates. You could switch the order of the templates within the stylesheet and the processor would give you the same result. Templates in a stylesheet may appear in any order and the order of their execution does not effect the execution of any other template.

Embedding a stylesheet

In the previous examples, you most likely used the EXMLPro editor to process the XML documents. Behind the scenes, the editor is programmatically connecting the stylesheet with the document. It uses an instance of the TXpXSLProcessor component and a filter component (e.g., HTML, RTF) to render the output document. The XmlObjModel property of the TXpXSLProcessor component is connected to a document object model containing the XML document. The Filter property of the TXpXSLProcessor component is then connected to the appropriate filter component. The editor tells the TXpXSLProcessor component to apply the styles in the stylesheet to the XML document and writes the resulting document to a file on disk. More details are provided in Chapter 7.

When you write applications using XMLPartner Professional, this is one of the methods you will use to associate a stylesheet with an XML document. One alternative to programmatic connection is to embed a stylesheet within an XML document. It is a good practice to put stylesheets in separate files because it allows you to re-use the stylesheets with a number of XML documents. However, you may encounter a situation where the XML document and its stylesheet must be self-contained.

To embed a stylesheet in an XML document, place the xsl:stylesheet element and its contents within the XML document. The stylesheet elements must be contained by the xsl:stylesheet element as they would be if they were in a stand-alone XSL stylesheet. However, xsl:stylesheet is no longer the outermost element of the document. Instead, the xsl:stylesheet element must be contained by the XML document's outermost element.

The following document, located in file embed.xml within subfolder Examples\Xmldocs of your XMLPartner Professional installation, uses an embedded stylesheet:


```
<?xml version="1.0"?>
<TopProducts>
  <Product company="Volcano" name="Throat lozenges"/>
  <Product company="SoftSpell" name="Car seat cushion"/>
  <Product company="Bunyon" name="Magicsoft toilet paper"/>
  <Product company="RedHat" name="Linux 7.1"/>
  <Product company="GoogleEye" name="Field binoculars"/>
  <Product company="Longshot" name="10-gauge shotgun"/>
  <Product company="Sparrow" name="stunt kite"/>
  <Product company="Plasticmaid" name="storage box"/>
  <Product company="Dullsworth" name="steak knife set"/>
  <Product company="Squinty" name="Eye scraper"/>

  <xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:text>StuffMart's </xsl:text>
    <xsl:value-of select="count(/TopProducts/Product)"/>
    <xsl:text> Most Sold Products</xsl:text>
    <br/>
    <ul>
      <xsl:for-each select="/TopProducts/Product">
        <li>
          <xsl:value-of select="@company"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="@name"/>
```

```
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>
  </xsl:stylesheet>
</TopProducts>
```

Processing the document with the EXMLPro editor produces an HTML document as shown in Figure 6.2.

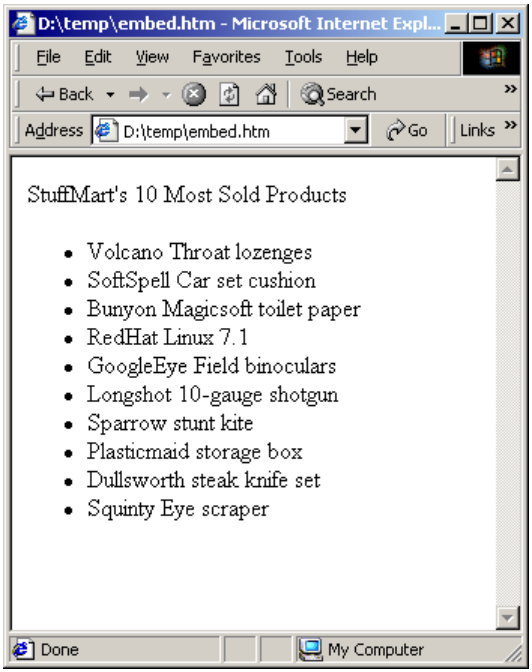


Figure 6.2: HTML document created with embedded stylesheet

Typically, an embedded stylesheet is given an id attribute and an <?xml-stylesheet?> processing instruction is included in the prolog of the XML document. However, XMLPartner Professional does not require the existence of <?xml-stylesheet?> in order to recognize and use the stylesheet.

Importing stylesheets

As seen in a previous example, StuffMart uses one stylesheet to format internal memorandums for printing and for display on an internal Web site. In general, information in internal memorandums is for internal use only. The information may not be disclosed to people outside the company. On occasion, the information in certain memorandums is very confidential and is not shared among the entire corporate staff. Confidential memorandums must be formatted in the same way as a regular memorandum with the exception that their title page must include the word Confidential at page top in bold, 18 point font.

At this point, you could choose to copy the BaseMemo.xsl stylesheet and modify it to suit this situation. However, this would mean two stylesheets must be modified if the format of the corporate memorandums was to ever change. A better alternative is to re-use BaseMemo.xsl stylesheet via the xsl:import instruction. The following stylesheet includes a template to meet the requirements of a confidential memo:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="BaseMemo.xsl">

  <xsl:template match="/">
    <fo:block font-size="18pt"
              font-weight="bold"
              text-align="centered">CONFIDENTIAL</fo:block>
    <xsl:apply-imports/>
  </xsl:template>
</xsl:stylesheet>
```

This stylesheet is located in your XMLPartner Professional installation as file Examples\Xmldocs\ConfMemo.xls. The ConfMemo stylesheet imports the BaseMemo stylesheet using the xsl:import element. The href attribute of xsl:import may be an absolute or relative reference to the stylesheet.

The template within the stylesheet matches the document root. A template matching the document root is also contained in the imported BaseMemo stylesheet. This could be considered a conflict. As described in the next section, the XSL processor handles conflicts by assigning priorities to templates. Templates in imported stylesheets are given lower priority than a corresponding template in the importing stylesheet. In this situation, the template in ConfMemo has a higher priority than the template in BaseMemo. However, the BaseMemo template still needs to be executed so that the memo has the correct heading. The template in ConfMemo causes the BaseMemo template to be executed by calling the xsl:apply-imports instruction. The XSL processor looks in the imported stylesheets for the highest priority template matching the same pattern as the current template.

XSLT's priority scheme makes the xsl:import element useful when you need to override a template in a stylesheet. If you had needed to add templates without changing existing templates then you could have used the xsl:include element. It is similar to xsl:import with the exception that included templates are assigned the same priority as templates in the including stylesheet.

Also notice this example uses only two levels of stylesheets. It is possible to nest stylesheets even deeper by having imported stylesheets import additional stylesheets. However, it is an error for a stylesheet to import itself either directly or indirectly.

Handling template collisions

When an XML document contains an element that matches more than one template in the stylesheet, XSLT defines rules for resolving conflicts. The first rule is that the XSL processor must determine which template or templates have the highest import precedence. As shown in the previous section, the xsl:import instruction allows one stylesheet to import another stylesheet. Templates within an imported stylesheet have a lower precedence than the templates in the importing stylesheet. If a stylesheet contains multiple imports then the import precedence increases from the first imported stylesheet to the last. In other words, the last imported stylesheet has the highest import precedence among its siblings.

To illustrate the idea of import precedence, look at Figure 6.3.

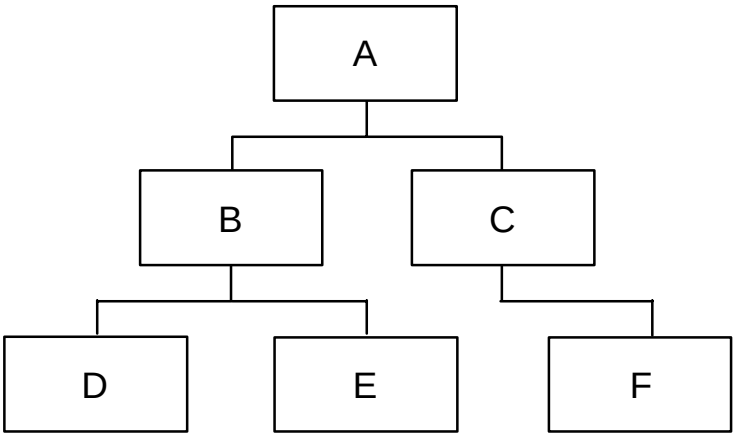


Figure 6.3: Import precedence diagram

Stylesheet A imports stylesheets B and C. Stylesheets B and C import D, E, and F. Stylesheet A has the highest import precedence because it directly or indirectly imports all of the other stylesheets. The order of import precedence, from highest to lowest, is A, C, F, B, E, D.

After the templates with highest import precedence have been chosen, the XSL processor compares the priority of each template. You can control the priority of a template by assigning its priority attribute a positive or negative number (e.g., 5, -1.0, 2.75). If a template does not have a priority attribute, the XSL processor assigns the template a default priority between 0.5 and –0.5 based upon the exactness of the template's pattern. In other words, a template that matches all nodes (e.g., '*') is assigned a lower priority than a template that matches a specific node (e.g., "invoices/invoice"). The rules for calculating the default priority are given in Section 5.5 of the XSLT specification.

If the XSL processor finds one template with the highest priority then it uses that template to process the element. If the XSL processor finds more than one template with the highest priority (e.g., 2 templates with priority 5) then the behavior depends upon the RaiseConflictError property of the TXpXSLProcessor component. The RaiseConflictError property defaults to False, which means the XSL processor uses the template that occurs last in the stylesheet. If you assign the value True to the RaiseConflictError property then the XSL processor raises an exception when it encounters this situation.

Programming a Stylesheet

XSLT includes a number of elements that provide functionality similar to that found in programming languages. For example, XSLT provides elements to support decision-making, global variables, and local variables. XSLT also provides a mechanism for invoking templates as though they were a procedure or subroutine. In this section, you will see how to use these elements of XSLT to construct stylesheets for Stuffmart.

Stuffmart accumulates a number of products in its inventory that it can't sell through normal means. To help rid itself of this inventory, Stuffmart has decided to create a "Special Sale" area on its Web site. The Special Sale area will feature these products at bargain prices. Information about the items is stored in an XML document. An example stylesheet named SaleStuf.xml is included in the Examples\Xmldocs subfolder of your XMLPartner installation. Each item has a description, a retail price, a sale price, a category code, and a flag indicating whether a picture of the item is available. In the following sections, you will construct the stylesheet that builds the HTML page.

Using conditional logic

XSLT includes two elements, xsl:if and xsl:choose, that allow you to embed conditional logic within a template rule. The xsl:if element is identical to an IF statement in most programming languages. However, it does not provide an IF..THEN capability. The xsl:choose element is similar to a CASE or SWITCH statement in other programming languages. It contains one xsl:when element for each possible option and an xsl:otherwise element that is invoked if none of the xsl:when elements are invoked. If an xsl:choose element contains only one xsl:when element and an xsl:otherwise element then it simulates an IF..THEN statement.

As described in the previous section, Stuffmart needs a stylesheet to generate a special sales page for its Web site. The stylesheet should transform item information into an HTML document. Your starting point is a stylesheet that lists the items in a table, as shown in the following code:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
  <table width="561" height="86" cellspacing="0">
    <tr align="center">
      <td width="258" height="8"
        style="border-left-style: solid;
               border-top-style: solid;
               border-bottom-style: solid">
        Description</td>
      <td width="65" height="8"
        style="border-top-style: solid;
               border-bottom-style: solid">
        Retail Price</td>
      <td width="71" height="8"
        style="border-top-style: solid;
               border-bottom-style: solid">
        Our Price</td>
      <td width="139" height="8"
        style="border-right-style: solid;
               border-top-style: solid;
               border-bottom-style: solid">
        Department</td>
    </tr>
    <xsl:apply-templates/>
  </table>
  </body>
  </html>
</xsl:template>

<xsl:template match="item">
  <tr>
    <td width="258" height="18">
      <xsl:value-of select="."/>
    </td>
    <td width="65" height="18" align="right">
      <xsl:value-of select="@retail"/>
    </td>
```

```
      <td width="71" height="18" align="right">
        <xsl:value-of select="@sale"/>
      </td>
      <td width="139" height="18" align="center">
        <xsl:value-of select="@category"/>
      </td>
    </tr>
  </xsl:template>

  </xsl:stylesheet>
```

The table created by this stylesheet is basic. One way to spice it up is to display a graphic for each item that has a picture available. The xsl:if element is ideally suited for this situation. The item template may be modified as follows to display a bitmap if the picture attribute is set to the value ‘y’:

6

```
  <xsl:template match="item">
    <tr>
      <td width="258" height="18"
          style="border-bottom-style: solid">
        <xsl:value-of select="."/>
        <xsl:if test="@picture='y'">
          <img border="0" src="file:../pic.bmp"
               width="16" height="15"/>
        </xsl:if>
      </td>
      <td width="65" height="18" align="right"
          style="border-bottom-style: solid">
        <xsl:value-of select="@retail"/>
      </td>
      <td width="71" height="18" align="right"
          style="border-bottom-style: solid">
        <xsl:value-of select="@sale"/>
      </td>
      <td width="139" height="18" align="center"
         style="border-bottom-style: solid">
        <xsl:value-of select="@category"/>
      </td>
    </tr>
  </xsl:template>
```

In this stylesheet, the xsl:if element tests the value of the picture attribute. If it is set to the value ‘y’ then an image tag is included in the output tree.

Sometimes an item's name doesn't adequately describe what is being sold or where the person might find it in the store. It would be helpful to display the item's department along with its name and price. Currently, the stylesheet displays only the category code. The xsl:choose element may be used to convert the category code into a department name. The following lines show the updated items template:

```
<xsl:template match="item">
  <tr>
    <td width="258" height="18"
        style="border-bottom-style: solid">
      <xsl:value-of select="."/>
      <xsl:if test="@picture='y'">
        <img border="0" src="file:../pic.bmp" width="16"
height="15"/>
      </xsl:if>
    </td>
    <td width="65" height="18" align="right"
        style="border-bottom-style: solid">
      <xsl:value-of select="@retail"/>
    </td>
    <td width="71" height="18" align="right"
        style="border-bottom-style: solid">
      <xsl:value-of select="@sale"/>
    </td>
    <td width="139" height="18" align="center"
       style="border-bottom-style: solid">
      <xsl:choose>
        <xsl:when test="@category='AP'">
        Appliances
        </xsl:when>
        <xsl:when test="@category='AU'">
        Auto
        </xsl:when>
        <xsl:when test="@category='GR'">
        Gardening
        </xsl:when>
        <xsl:when test="@category='SH'">
        Shoes
        </xsl:when>
        <xsl:when test="@category='TY'">
        Toys
        </xsl:when>
        <xsl:otherwise>
          Miscellaneous
        </xsl:otherwise>
```

```
      </xsl:choose>
    </td>
  </tr>
</xsl:template>
```

An xsl:choose statement contains one or more xsl:when elements and an optional xsl:otherwise element. When the template is invoked, the XSL processor evaluates each xsl:when element until it finds one that evaluates to True. The body of the matching xsl:when element is then output to the result tree. If none of the xsl:when elements evaluates to True and an xsl:otherwise element is found, the XSL processor outputs the body of the xsl:otherwise element to the result tree. In this situation, if the template does not recognize the product's category then the template defaults to the Miscellaneous category.

The completed stylesheet for this section is stored in the file Sale1.xsl in the Examples\Xmldocs subfolder of your XMLPartner installation.

Calling templates

Up to this point, the stylesheet templates have been invoked as necessary by the XSL processor. However, it is possible to call a template from another template using the xsl:call-template element. This is analogous to a procedure or subroutine call in other programming languages. It allows you to create reusable templates that may be imported into any number of stylesheets. Called templates must have a name and may accept zero or more parameters using the xsl:param element.

Continuing the Stuffmart special sales example, the HTML document produced by the stylesheet leaves the numbers in the same form as they are pulled from the XML document. They may have been entered into the XML document as an integer or decimal number. Either way, the number should be reformatted so that it is consistently displayed as a price in the HTML document.

To solve the problem, the stylesheet needs a template for formatting numbers as dollar amounts. The template is as follows:

```
<xsl:template name="FormatPrice">
  <xsl:param name="Price" select="0"/>
  <xsl:value-of select="format-number($Price,'###.00')"/>
</xsl:template>
```

Because the template has a name attribute, it may be called from other templates. It must be passed a dollar value to be formatted, so it also includes an xsl:param element. If a template calls FormatPrice, it should include an xsl:with-param element having the name "Price" and

the value that is to be formatted. Otherwise, the “Price” template will use the default price of 0 specified for the xsl:param element. A version of the item template that uses FormatPrice is as follows:

```
<xsl:template match="item">
  <tr>
    <td width="258" height="18"
        style="border-bottom-style: solid">
      <xsl:value-of select="."/>
      <xsl:if test="@picture='y'">
        <img border="0" src="file:../pic.bmp"
             width="16" height="15"/>
      </xsl:if>
    </td>
    <td width="65" height="18" align="right"
        style="border-bottom-style: solid">
      <xsl:call-template name="FormatPrice">
        <xsl:with-param name="Price" select="@retail"/>
      </xsl:call-template>
    </td>
    <td width="71" height="18" align="right"
        style="border-bottom-style: solid">
      <xsl:value-of select="@sale"/>
    </td>
    <td width="139" height="18" align="center"
       style="border-bottom-style: solid">
      <xsl:choose>
        <xsl:when test="@category='AP'">
        Appliances
        </xsl:when>
        <xsl:when test="@category='AU'">
        Auto
        </xsl:when>
        <xsl:when test="@category='GR'">
        Gardening
        </xsl:when>
        <xsl:when test="@category='SH'">
        Shoes
        </xsl:when>
        <xsl:when test="@category='TY'">
        Toys
        </xsl:when>
        <xsl:otherwise>
          Miscellaneous
        </xsl:otherwise>
     </xsl:choose>
```

```
      </td>
    </tr>
  </xsl:template>
```

In this example, the stylesheet calls a named template that accepts one parameter. It is also legal to create a named template that doesn't have any parameters or that has multiple parameters. The stylesheet for this section is saved in the Sale2.xsl file of the Examples\Xmldocs subfolder.

Using variables

XSLT supports the use of variables global to a stylesheet as well as variables local to a template. Global variables are declared as child elements of the xsl:stylesheet element while local variables are declared within the bodies of template rules. In most programming languages, global variables can be changed during the execution of the program. In XSLT, the value of a global variable is fixed when the stylesheet is first executed and it may not be changed after that point.

Continuing the Stuffmart example, the current stylesheet dumps all items to the HTML document. As the number of items grows, this will become unwieldy. The items could be output by category. The first step is to declare a global variable as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:variable name="ItemCategory" select="'AU'"/>
...
</xsl:stylesheet>
```

The xsl:variable element defines a global variable named ItemCategory and assigns it a value using its select attribute. In this situation, the variable is assigned the string value "AU." Variables can also be assigned a Boolean, numeric, or node-set value. If no select attribute is specified, the value can be assigned using the body of the xsl:variable as shown in the following example:

```
<xsl:variable name="ItemCategory">AU</xsl:variable>
```

The global variable may then be referenced in the match attribute of the item template as shown in the following example:

```
<xsl:template match="item[@category=$ItemCategory]">
...
</xsl:template>
```

The XMLPartner TXpXSLProcessor class provides the methods GetVariable and SetVariable for retrieving and setting the values of global variables, respectively. The Examples\Delphi\GloblVar subfolder of your XMLPartner installation contains an example

that controls the ItemCategory variable within the stylesheet, allowing you to programmatically generate one HTML document per item category. The final stylesheet for this section is saved in the Sale3.xsl file of the Examples\Xmldocs folder.

Local variables are declared using the xsl:variable element and values may be specified in the same way as for global variables. However, the scope of a local variable is limited to the template in which it is declared. If the local variable is declared in the middle of a template, the variable is visible only to those elements occurring after its declaration within the template body. The XMLPartner XSL processor does not provide an interface to programmatically access the values of local variables.

Summary

This chapter has guided you through a small portion of the capabilities present in XSL. You have learned how to declare a stylesheet, how to create a template that places information into a result tree, and how to embed a stylesheet within an XML document. You also learned how to use conditional logic within a stylesheet, call a template as though it were a subroutine, and how to use global variables. This should be enough to get you started with XSL. To learn more about the XSLT elements and XSL formatting objects supported by XMLPartner Professional, refer to the reference material in the following chapters. Even more information and examples can be found in the references cited in Chapter 1.

Chapter 7: XSL Processor & Filter Reference

XMLPartner Professional's XSL components are designed to transform XML documents into new documents with layouts and formatting for presentation. Using the XSL components and an XSL stylesheet, an XML document can easily be converted into HTML, XML, RTF, or a printed document. This chapter details these components and demonstrates how to use them to transform XML documents into new documents and prepare them for presentation with minimal time and effort.

The XSL components include an XSL processor and filters to generate specific types of documents. The XSL processor applies the transformation instructions from the stylesheet to the XML document to produce a result tree. A *filter* is a component that maintains the result tree and applies the formatting instructions from the stylesheet to produce output in a specific format. The HTML filter generates HTML documents. The RTF filter generates documents in Microsoft's Rich Text Format (RTF). The XML filter generates new XML documents. The Print filter is a visual control that renders a print preview that can be printed.

The Result Tree

To understand how the XMLPartner Professional XSL processor and filter components work together, it is important to understand what a result tree is and how it's formed. There are two distinct parts of XSL: XSL-Transformations (XSLT) and XSL-Formatting Objects (XSL-FO). XSLT is the part that generates the result tree.

The transformation takes place as the XSL processor applies the commands in the XSL stylesheet to the XML document. The stylesheet commands tell the processor what data to insert into the result tree and how to format the data. The data may come directly from the source tree (XML document), be calculated from the data in the source tree, or generated directly by the stylesheet. The XSL stylesheet commands and formatting object commands are described in Chapters 8 and 9, respectively.

As the processor is applying the stylesheet commands, it raises events that send the results to a maintainer that is responsible for maintaining the result tree. While a filter is usually the maintainer, it is possible for an application to retrieve the events and maintain the result tree directly. There are examples of both situations provided later in this chapter.

The processor also recognizes the Formatting Objects part of XSL. Formatting object commands are used to apply formatting information to the result tree. As formatting object commands are encountered, the processor raises events to notify the maintainer it needs to apply the formatting to the current section of data. There are formatting object commands to specify page layout, group blocks of data together, create tables, select background color and graphics, and select fonts. The logic required to apply these commands is specific to the type of output being generated by the maintainer. In other words, changing the background color is done differently in an HTML document than an RTF document. For this reason, the maintainer is also responsible for applying the format object commands for its designated form of output. The formatting object commands and properties are explained in Chapter 9.

With the information about how XMLPartner's XSL processor and filters work together, the next step is to see how the components are used in an actual application. The next section, "Using the XSL Components", demonstrates how to use the components in an application.

Using the XSL Components

This section demonstrates how to use the XSL processor and filter components in a step-by-step manner. Steps 2 and 6 will differ depending on which, if any, filter component is being used. If a filter isn't being used, read the "Using the XSL processor without a filter" section on page 244 after reading the following steps:

1. Place a TXpObjModel on the form. This component will contain an XML document to which an XSL stylesheet will be applied.
2. Place a TXpXSLProcessor on the form. This component will contain an XSL stylesheet that is applied to the XML document. This component is also responsible for applying the stylesheet to the XML document.
3. Place a filter on the form. Each of the filter components is responsible for generating output in a specific form as described later in this chapter. Note that a filter isn't actually required, but most people will want to use one. The "Using a processor without a filter" section demonstrates a situation where a filter isn't required. Now that the components are in place it's time to connect them.
4. Set the TXpXSLProcessor's XMLObjModel property to the TXpObjModel component you placed on the form. This tells the processor which XML document to apply the stylesheet to. This property can be set at design time using the Object Inspector or at run time using code similar to the following line:

   ```
   XSLProcessor.XMLObjModel := DOM;
   ```

5. Set the TXpXSLProcessor's Filter property to the filter you placed on the form. This tells the processor where to send the result tree and formatting object commands. This property can be set at design time using the Object Inspector or at run time using code similar to the following line:

   ```
   XSLProcessor.Filter := FilterHTML;
   ```

   Now all the components are connected and it's time to setup the XML document and XSL stylesheet.

6. Prepare the XML document. Before the XSL processor can apply a stylesheet to an XML document, the XML document must be parsed. To parse the XML document, call the XpObjModel's LoadDataSource or LoadMemory method. If using LoadDataSource, pass the file name to an XML document as shown in the following source code lines:

   ```
   if (not XpObjModel.LoadDataSource('C:\Test.xml')) then
     ShowMessage('Unable to parse XML document');
   ```

If using LoadMemory, pass it a buffer containing the XML document as shown in the following example:

```
if (not XpObjModel.LoadMemory(Buff[1], Length(Buff))) then
  ShowMessage('Unable to parse XML document');
```

For additional details about the LoadDataSource and LoadMemory functions, see Chapter 4.

7. Assign the XSL processor an XSL stylesheet. Like the XML document in step 3, the stylesheet can be loaded from a file or from memory. To load it from a file use TXpXSLProcessor's StyleURL property at design time or using code similar to the following example at run time:

```
XSLProc.StyleURL := 'C:\Test.xsl';
```

If the stylesheet is contained in memory as a string, WideString, or DOMString, load the stylesheet as shown in the following line of code:

```
XSLProc.StyleData := MyStyle;
```


8. Apply the XSL stylesheet. Now that all the components are setup, the XML document is parsed, and the stylesheet is assigned, the processor is ready to apply the stylesheet to the XML document. Call the processor's ApplyStyle method to apply the stylesheet to the XML document as shown in the following piece of code:

```
if (not XSLProc.ApplyStyle) then
  ShowMessage('Unable to apply stylesheet');
```

9. Retrieve the processed document from the filter. If the application has made it this far, it's time to see the results of the processing. The results differ somewhat depending on the type of filter used.

Except for the Print Filter, all of the documents can be saved to disk using the following code:

```
Filter.SaveToFile('NewDoc.txt');
```

Using a Print Filter, the document is displayed on the component after calling ApplyStyle. The Print Filter's document can also be printed using the following line of code:

```
PrintFilter.Print;
```

Using an HTML Filter, the document can be retrieved using the component's HTMLDocument property.

```
Memo.Text := HTMLFilter.HTMLDocument;
```

The RTF Filter's document can be retrieved much like the HTML document. After saving the RTF document to disk, it can be viewed with formatting in a TRichEdit component:

```
RTFFilter.SaveToFile('C:\MyRTF\NewRTF.rtf');
RichEdit.PlainText := False;
RichEdit.Lines.LoadFromFile('C:\MyRTF\NewRTF.rtf');
```

The raw (unformatted) RTF can be viewed in a TMemo component using the following line of code:

```
Memo.Text := RTFFilter.RTFDocument;
```

The XML Filter's document is also retrieved in a similar manner:

```
XpObjModel2.LoadMemory(
  XMLFilter.XMLDocument[1], Length(XMLDocument));
```

There is a small project called XSLDemo in a subdirectory under XMLPartner's Examples directory that demonstrates the components' property values and event handlers in a basic application. The project uses a Print filter to preview and print processed XML documents.

TXpXSLProcessor Component

The XSL processor component, TXpXSLProcessor, contains the logic to apply an XSL stylesheet to an XML document. It is responsible for applying the transformation instructions from a XSL document to a XML document to produce the result tree. The processor is also responsible for notifying a maintainer (usually a Filter) when elements, attributes, text, and formatting objects are found so the user can format the result tree appropriately.

A filter will normally handle TXpXSLProcessor's events. This means most people using the XSL components won't be interested in its events unless they're creating a custom Filter. The events are used to notify a user what is being encountered as the stylesheet is processed. This allows a user to generate the transformed and formatted document as specified by the stylesheet. If the supplied Filters don't produce the needed type of output, events can be handled directly or by a custom filter. See the Filters section of this chapter for more information about creating a custom filter.

Using the XSL processor without a filter

In most cases one of XMLPartner's Filters will be used to maintain the result tree generated by the XSL processor, but it's not required. The modular design of the components allows an application to handle the processor's events, maintain the result tree, and apply formatting information directly. This section demonstrates processing the events directly to convert a specific type of XML document into a comma delimited file. The completed project is called NoFilter and is located in a subdirectory under XMLPartner's Examples directory.

The scenario

The scenario for this project is that daily weather reports are received as XML documents, but they need to be input into an antiquated system that doesn't recognize XML. The system expects input in a comma-delimited format. An example of the XML formatted weather reports is in XMLPartner's directory. The following is a code snippet from that document:

```
<?xml version="1.0"?>
<WEATHERREPORT>
    <STATE NAME="California">
        <CITY NAME="Los Angeles">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="31" F="87"/>
            <LOW C="18" F="65"/>
            Partly cloudy
        </CITY>
        …
        <CITY NAME="Truckee">
            <SKIES VALUE="RAIN"/>
            <HI C="32" F="89"/>
            <LOW C="11" F="52"/>
            Scattered thunderstorms
        </CITY>
    </STATE>
    <STATE NAME="New Jersey">
        <CITY NAME="Newark">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="36" F="97"/>
            <LOW C="21" F="71"/>
            Partly sunny, breezy and humid
        </CITY>
        …
    </STATE>
    …
</WEATHERREPORT>
```

The XSL stylesheet

The following XSL stylesheet extracts the required information from the XML formatted weather report:

```
<?xml version="1.0"?>
  <!--XML declaration-->
<xsl:stylesheet>
  <!--Start stylesheet-->
  <xsl:template match="/STATE">
    <!--Get all STATE elements-->
    <xsl:for-each select="CITY">
      <!--Iterate STATE’s CITY elements-->
      <xsl:element name="city">
        <!--Mark start of new record-->
        <xsl:value-of select="@NAME"/>
          <!--Send CITY's NAME to result-->
        <xsl:value-of select="../@NAME"/>
          <!--Send STATE's NAME to result-->
        <xsl:value-of select="HI/@C"/>
          <!--Send CITY's HI temp to result-->
        <xsl:value-of select="LOW/@C"/>
          <!--Send CITY's LO temp to result-->
        <xsl:value-of select="."/>
          <!--Send forecast text to result-->
      </xsl:element>
        <!--Mark the end of the record-->
    </xsl:for-each>
      <!--End of the CITY elements-->
  </xsl:template>
    <!--Release STATE elements-->
</xsl:stylesheet>
  <!--End stylesheet-->
```

The stylesheet selects each city element in each state element contained in the weather report. For each city, it creates an element named city that signals the start of each new record. After that, it passes the name of the current city, name of the current state, the city’s high and low temperature, and the text containing the current day forecast.

The conversion

Each city’s data must be converted to a file that contains that city’s information on a separate line and a comma must separate each piece of information. By creating simple handlers for only five of the XSL processor’s events, this becomes a quick and easy task.

Following the flow of the stylesheet, the first task is to recognize that a new report is being processed. As each report is processed, the application should clear out all data from previously processed reports. The processor notifies the application when a new report is being processed when it receives the OnApplyStyleStart event. The following is the handler for this event:

```
procedure TfrmMain.XslProcApplyStyleStart(Sender : TObject);
begin
  {Ensure data file is empty}
  CSVFile.Clear;
end;
```

As the processor applies the stylesheet to the weather report, the next event it will raise is OnElementStart. This tells the application that the next data received is for a new city, so the application needs to clear out any variables used for processing the previous city. Following is the handler for that event:

```
procedure TfrmMain.XSLProcElementStart(
  oOwner : TObject; sValue : WideString);
begin
  {Clear out the current record to prepare for new city's data}
  CurrentRecord := '';
end;
```

As each piece of the city's data is sent to the result tree, the processor will raise an OnText event containing the text. This is how the program actually builds the comma-delimited record for each city's forecast. The following is the OnText event handler that builds each record:

```
procedure TfrmMain.XSLProcText(
  oOwner: TObject; sValue: WideString);
begin
  {Is this the first field in the record?}
  if (CurrentRecord = '') then
    {Yes. No need to separate new field with a comma.}
    CurrentRecord := sValue
  else
    {No. Add comma delimiter and new data to the current record.}
    CurrentRecord := CurrentRecord + ',' + sValue;
end;
```

After all the city's data has been sent to the application, the processor will raise the OnElementEnd event that signals the end of the current city's record. This is the application's cue to add the current record to the object containing the new file. Following is the OnElementEnd event handler that does just that:

```
procedure TfrmMain.XSLProcElementEnd(
  oOwner : TObject; sValue : WideString);
begin
  {Add the current record to the new file.}
  CSVFile.Lines.Add(CurrentRecord);
end;
```

The last event the application needs to watch for is the OnApplyStyleEnd. This signals the application that all of the records in the weather report have been processed. At this point, the program can save the new CSV file to disk as shown in the following OnApplyStyleEnd handler:


```
procedure TfrmMain.XslProcApplyStyleEnd(Sender : TObject);
begin
  {All records processed. Save CSV file to disk.}
  CSVFile.Lines.SaveToFile('DailyWeather.txt');
end;
```

The result

After executing the program with the previous event handlers, the program converts the XML formatted weather report into a comma-delimited file of city forecast records without using a filter. It handled the XSL processor's events and maintained the result tree directly. The following is the contents of the new comma-delimited file:

```
Los Angeles,California,31,18,Partly cloudy
Sacramento,California,36,17,Sunny and hot
San Diego,California,26,19
San Fransisco,California,26,14,Partly cloudy and humid
Truckee,California,32,11,Scattered thunderstorms
Newark,New Jersey,36,21,Partly sunny, breezy and humid
Trenton,New Jersey,32,18,Partly cloudy and humid
Princeton,New Jersey,33,20,Thundershowers
Seattle,Washington,20,15,Raining on and off throughout the day
Yakima,Washington,23,14,Partly sunny after morning clouds
Redmond,Washington,2,-7,Snowstorms in the afternoon
```

Hierarchy

TComponent (VCL)

TXpXSLProcessor (XpXSLPrc)

Properties

Filter	RaiseErrors	StyleURL
IgnoreCase	StyleData	XpObjModel

Methods

ApplyStyle	Errors	SetVariable
ClearVariables	GetVariable	

Events

OnApplyStyleEnd	OnElementEnd	OnProcessingInstruction
OnApplyStyleStart	OnElementStart	OnQueryFilter
OnAttribute	OnFoProperty	OnResolveElement
OnCDATASection	OnFormatObjectEnd	OnResolveElementCode
OnComment	OnFormatObjectStart	OnText

ApplyStyle — method

```
function ApplyStyle : Boolean;
```

Applies an XSL stylesheet to an XML document.

Applying a stylesheet to an XML document creates a result tree. The stylesheet is the stylesheet assigned to the processor's StyleData or StyleURL property. The XML document is the document loaded into the TXpObjModel that is assigned to the processor's XMLObjModel property. As stylesheet is applied to the document, the result tree and formatting instructions are passed to the maintainer. See the introduction to this chapter for more information about this process.

ApplyStyle returns True if the processor applies the stylesheet without errors and False if it encounters a problem. If ApplyStyle encounters errors, they can be retrieved using the Errors property. A common reason that ApplyStyle returns False is not assigning an instance of TXpObjModel to the XpObjModel property. If ApplyStyle returns True and there is a filter assigned to the filter property, the resulting document is ready to be viewed, saved, or printed.

See also: Errors, Filter, StyleData, StyleURL, XpObjModel

ClearVariables — method

```
procedure ClearVariables;
```

Clears the global variables used when applying an XSL stylesheet to an XML document.

When using a TXpXSLProcessor to apply multiple stylesheets, this method should be called before each stylesheet is applied to clear all global variables used by previous stylesheets, as shown in the following example:

```
var
i : Integer;
begin
  Assert(XMLList.Count = XSLList.Count);
  for i := 0 to Pred(XSLList.Count) do begin
    if (DOM.LoadDataSource(XMLList.Strings[i])) then begin
      XSLP.StyleURL := XSLList.Strings[i];
      XSLP.ClearVariables;
      if (not XSLP.ApplyStyle) then
        ShowMessage(format('Couldn''t apply %s to %s',
          [XSLList.Strings[i], XMLList.Strings[i]]))
```

```
        else
          HTMLFilter.SaveToFile('Report' + IntToStr(i) + '.htm');
      end else
        ShowMessage('Error: ' + DOM.GetErrorMsg(0));
    end;
  end;
```

See also: GetVariable, SetVariable

Errors — run-time property

```
property Errors : TStringList
```

Retrieves the errors encountered during a call to ApplyStyle.

If ApplyStyle returns False, retrieve the value of this property to find out what errors were encountered. The following code snippet is an example of using this property:

```
if (not XSLProc.ApplyStyle) then
  memResults.Lines.Assign(XSLProc.Errors);
```

See also: ApplyStyle

Filter — property

```
property Filter : TXpFilterBase
```

Identifies the XMLPartner filter component used to produce output of a specific type.

XMLPartner Professional provides filters to produce four different types of output from an XML document using an XSL stylesheet: TXpFilterHTML, TXpFilterPrint, TXpFilterRTF, and TXpFilterXML. TXpFilterHTML produces an HTML document, TXpFilterPrint produces a document that can be previewed or printed. TXpFilterRTF produces an RTF document. TXpFilterXML produces a new XML document. Custom filters can be derived from the TXpFilterBase class and assigned to this property.

Most people will use a filter to produce the desired output, but it is not required. Setting the filter property routes the XpXSLProcessor's events to the predefined event handlers of the Filter. The filter produces a document in its designated form. If the filter property isn't assigned, the events can be handled directly by an application. See the Using the XSL processor without a filter section earlier in this chapter for additional information.

See "Filters" on page 262 for information about each of the installed filters and information about creating custom filters.

See also: ApplyStyle

GetVariable — method

```
function GetVariable(sName : DOMString) : DOMString;
```

```
DOMString = WideString;
```

Retrieves the value of a global variable used during processing.

sName is the name of the global variable to retrieve. If the variable is found, the function returns the value of the variable. If the variable isn't found, an empty string is returned.

See also: ClearVariable, SetVariable

IgnoreCase — property

```
property IgnoreCase : Boolean
```

Default: False

Specifies whether patterns used in transformations are case sensitive.

XML is case sensitive as per the XML 1.0 specification. Setting this property to True allows matching patterns without regard to the case of the element names. Using the following XML document:

```
<?xml version="1.0"?>
<WEATHERREPORT>
    <STATE NAME="California">
        <CITY NAME="Los Angeles">
          Partly cloudy
        </CITY>
  </STATE>
</WEATHERREPORT>
```

The following XSL stylesheet would not produce any results if this property were False because the match attribute on the third line does not match any element names in the XML document above:

```
<?xml version="1.0"?>
<xsl:stylesheet>
  <xsl:template match="/state">
    <xsl:for-each select="CITY">
      <xsl:element name="city">
        <xsl:value-of select="@NAME"/>
        <xsl:value-of select="../@NAME"/>
      </xsl:element>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Changing the IgnoreCase property to True would allow the stylesheet to match an element and send the data to the result tree.

OnApplyStyleEnd — event

```
property OnApplyStyleEnd : TNotifyEvent
```

Defines an event handler that is called after XSL processing is completed.

This is the last event raised when ApplyStyle is called. If an error is encountered and RaiseErrors is True, this event will not be raised. The event is raised when an error occurs and RaiseErrors is False.

See also: ApplyStyle, RaiseErrors

OnApplyStyleStart — event

```
property OnApplyStyleStart : TNotifyEvent
```

Defines an event handler that is called before XSL processing begins.

This event is raised after the ApplyStyle method is called and the OnQueryFilter event is raised.

See also: ApplyStyle, OnApplyStyleEnd, OnQueryFilter, StyleData, StyleURL

OnAttribute — event

```
property OnAttribute : TXpAttributeEvent
```

```
TXpAttributeEvent = procedure(
  oOwner : TObject; sName, sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called when an attribute is sent to the result tree.

This event is raised for xsl:attribute commands. oOwner is the XpXSLProcessor raising the event. sName is the name of the attribute and sValue is its value.

See also: OnElementEnd, OnElementStart, OnFoProperty

OnCDATASection — event

```
property OnCDATASection : TXpCDATAEvent
```

```
TXpCDATAEvent = procedure(oOwner : TObject; sValue : DOMString;
  bReplace : Boolean) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called when a CDATA section is passed to the result tree.

oOwner is the XpXSLProcessor raising the event. sValue is the data contained in the CData section. bReplace indicates if special characters in sValue should be replaced. The HTML filter uses this event handler to replace special characters (& and <) with their entity equivalents (&amp and &lt respectively) to prevent producing ambiguous HTML code.

OnComment — event


```
property OnComment : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called for each comment sent to the result tree.

Comments are sent to the result tree using the xsl:comment stylesheet command. oOwner is the XSL processor raising the event. sValue is the text of the comment. This and other stylesheet commands are described in Chapter 8.

OnElementEnd — event

```
property OnElementEnd : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called after each element is sent to the result tree.

Elements are sent to the result tree using the xsl:element or xsl:pass-through instructions. This event is raised after the commands have been processed. See Chapter 8 for more information about these and other XSL commands. Result trees are described in “The Result Tree” on page 240.

oOwner is the XpXSLProcessor raising the event and sValue is the name of the element.

See also: OnAttribute, OnElementStart

OnElementStart event

```
property OnElementStart : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called before each element is sent to the result tree.

Elements are sent to the result tree using the xsl:element command. See Chapter 8 for more information about these and other XSL commands. Result trees are described in "The Result Tree" on page 240.

oOwner is the XpXSLProcessor raising the event and sValue is the name of the element.

See also: OnAttribute, OnElementEnd

OnFoProperty event

```
property OnFoProperty : TXpPropertyEvent
```

```
TXpPropertyEvent = procedure(oOwner : TObject;
  wCode  : Integer; sName, sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called for each formatting object property.

Formatting object properties are used to specify details like font, color, and page size. Properties appear as attributes of the formatting object elements, but are treated differently by the XSL processor since the maintainer of the result tree may need to apply the formatting details to its document.

Filter implementations need to handle this event and apply the property value if it applies to the Filter's output document type. For instance the provided HTML filter ignores the keep-with-next property since HTML documents are contained in a single scrollable page, but the RTF filter does capture the property to ensure the sections are kept together on the same page as requested.

Each of the formatting object properties is explained in Chapter 9.

oOwner is the XSL processor raising the event. wCode is the numeric code for the formatting object property. These numeric codes are defined in the Appendix. sName is the name of the property or attribute and sValue is the value of the property or attribute.

See also: OnAttribute, OnFormatObjectEnd, OnFormatObjectStart

OnFormatObjectEnd — event

```
property OnFormatObjectEnd : TXpFormatObjectEndEvent
```

```
TXpFormatObjectEndEvent = procedure(
  oOwner  : TObject; aFOType : Integer) of object;
```

Defines an event handler that is called after each formatting object command is processed.

The OnFormatObjectStart reference contains more information about how formatting objects are processed.

See also: OnFoProperty, OnFormatObjectStart

OnFormatObjectStart — event

```
property OnFormatObjectStart : TXpFormatObjectStartEvent
```

```
TXpFormatObjectStartEvent = procedure(oOwner : TObject;
  aFOType : Integer; const aData : array of const) of object;
```

Defines an event handler that is called before each formatting object command is processed.

If implementing a custom filter that is responsible for formatting output (such as a PDF Filter) handle this event and check for formatting instructions that the filter needs to implement. For instance, a PDF filter implementation needs to check for aFOType of FOP_BACKGROUND_IMAGE, among other aFOTypes, and apply the image to the background of the current section of the result tree.

oOwner is the XpXSLProcessor raising the event. aFOType is the type of formatting object being raised. The formatting object types are defined in the Appendix. aData contains the set of required parameters needed for the type of formatting object. The required parameters are described with each formatting object's description in Chapter 9.

See also: OnFoProperty, OnFormatObjectEnd

OnProcessingInstruction event

```
property OnProcessingInstruction : TXpAttributeEvent
```

```
TXpAttributeEvent = procedure(
  oOwner : TObject; sName, sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called when a processing instruction node is encountered.

The XSL processor uses this event to pass processing instructions from a stylesheet to a filter. The processing instructions are created in the XSL stylesheet using the xsl:processing-instruction command. See Chapter 8 for more information on this XSL instruction and others.

oOwner is the XSL object raising the event. sName is the name of the processing instruction and sValue is the value of the processing instruction.

OnQueryFilter event

7

```
property OnQueryFilter : TXpQueryFilterEvent
```

```
TXpQueryFilterEvent = procedure(
  oOwner : TObject; var sName : string) of object;
```

Defines an event handler that is called when querying for the name of a filter.

Custom filters should assign a unique name to their protected FDisplayName attribute in their constructor and override this handler to return a display name for the filter. This event is called after ApplyStyle is called and before OnApplyStyleStart.

oOwner is the XSL processor raising the event. The Filter's name is assigned to sName in the filter's handler.

See also: ApplyStyle, OnApplyStyleStart

OnResolveElement — event

```
property OnResolveElement : TXpResolveElementEvent
```

```
TXpResolveElementEvent = procedure(oOwner : TObject;
  wCode : Integer; var sTag : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that allows changing an XSL command name to another name.

oOwner is the XSL processor raising the event. wCode is the XSL tag code. The tag codes are listed in the Appendix. sTag contains that default command name that can be reassigned.

See also: OnResolveElementCode

OnResolveElementCode — event

```
property OnResolveElementCode : TXpResolveElementCodeEvent
```

```
TXpResolveElementCodeEvent = procedure(oOwner : TObject;
  sElement : DOMString; var wCode : Integer) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called for each element in the XSL stylesheet.

One use of this event is to replace an unrecognized XSL tag code with a recognized one. For instance, some old Microsoft products used xsl to identify a stylesheet. Since xsl isn't a recognized element name, it is assigned a tag code of UNKNOWN. The event can be used to change the UNKNOWN element code with a tag code of TAG_STYLESHEET to allow proper processing of the unrecognized element.

oOwner is the XSL processor raising the event. sElement is the name of the element tag found and wCode a code showing how the code was recognized. The Appendix contains a list of tag codes.

See also: OnResolveElement

OnText — event

```
property OnText : TXpValueEvent
```

```
TXpValueEvent = procedure(
  oOwner : TObject; sValue : DOMString) of object;
```

```
DOMString = WideString;
```

Defines an event handler that is called for each text element sent to the result tree.

Text is sent to the result tree using the following XSL stylesheet commands: xsl:text, xsl:value-of, xsl:attribute, xsl:processing instruction, xsl:comment, and xsl:number. These and other stylesheet commands are described in Chapter 8. The result tree is described in "The Result Tree" on page 240.

oOwner is the processor raising the event and sValue is the text being passed to the filter.

RaiseErrors — property

```
property RaiseErrors : Boolean
```

Default: False

Specifies whether the XSL processor is to raises errors to the application.

Setting this value to True causes errors to be raised to the application. If this property is False, the processor traps any errors.

See also: OnApplyStyleEnd

SetVariable — method

```
procedure SetVariable(sName, sValue : DOMString);
```

```
DOMString = WideString;
```

Overrides the value of a global variable used during processing.

This can be useful for overriding the value of variables established using the xsl:variable stylesheet command in the event handlers of a custom filter. sName is the name of the variable and sValue is its value.

See also: ClearVariables, GetVariable

StyleData — run-time property

```
property StyleData : DOMString
```

```
DOMString = WideString;
```

Specifies an in memory stylesheet to apply to an XML document.

Use this property to set a collection of stylesheet commands that are in memory. If the commands are contained in a file on a local disk drive, an ftp site, or http site, use the StyleURL property.

Setting the StyleData property removes any previous stylesheet commands assigned using the StyleData or StyleURL properties. Retrieving the value of StyleData returns the current set of style commands whether they were assigned with StyleData or StyleURL property.

See also: ApplyStyle, StyleURL, XpObjModel

StyleURL — property

```
property StyleURL : DOMString
```

```
DOMString = WideString;
```

Specifies an XSL stylesheet to apply to an XML document.

Use this property to set a stylesheet that is contained in a file on a local disk, ftp site, or http site. Use the StyleData property if the commands are in memory.

Setting the StyleURL property removes any previous stylesheet commands assigned using the StyleData or StyleURL properties. Retrieving the value of StyleURL returns the current set of style commands whether they were assigned with StyleData or StyleURL property.

Following are some examples of valid StyleURL assignments:

```
XSLProc.StyleURL := 'n:\xml\tugs\manifest.xsl';
XSLProc.StyleURL := 'file://d:\breeding\lineage.xsl';
XSLProc.StyleURL := 'http://www.myFreeBeer.com/supports.xsl';
XSLProc.StyleURL := 'ftp://ftp.weightloss.com/badFoodIdeas.xsl';
```

See also: ApplyStyle, StyleData, XpObjModel

```
property XpObjModel : TXpObjModel
```

Identifies a TXpObjModel containing the parsed XML document to which the stylesheet is applied.

The TXpObjModel object contains the parsed XML document to which the stylesheet is applied. The TXpObjModel's LoadMemory or LoadDataSource method should be called before calling the XSL processor's ApplyStyle method.

Unless using an XML document that contains embedded XSL, the XpObjModel property must be assigned before calling ApplyStyle. If using embedded XSL, assign the document containing the XML and XSL data using the StyleData or StyleURL property.

See also: ApplyStyle, StyleData, StyleURL

Filters

Filters are used to maintain and apply formatting instructions to the result trees generated by an XSL processor. Each filter component is designed to produce a specific type of output. The HTML filter produces HTML documents. The RTF filter produces Microsoft's Rich Text Format (RTF) documents. The XML filter produces XML documents. The Print filter is a visual component that displays a printed document that can be sent to a printer.

The filters accomplish their assigned tasks by catching the XSL processor's events and handling the data and instructions as required by its form of output. All of the filters maintain a result tree, but not all of them have to respondto the formatting object events. For instance, XML documents are not designed to display information—that's why stylesheets were invented. An XML document's purpose is to store information. Because of this, the XML filter ignores the processor's formatting object events.

The rest of this chapter explains the filters in detail. First, the base filter class is explained. The next four sections detail the specifics for each of the filter components. The last section explains how to implement a custom filter.

TXpFilterBase Class

TXpFilterBase contains the core logic for all the filters, but it is not a component and should not be used directly. Because all of the filter components are responsible for maintaining the result tree, they all have some common functionality. This common functionality is contained in the base filter class. All of the filter components provided by XMLPartner Professional inherit from this class. Deriving custom filters from this class will also make the implementor's job much easier.

The rest of this section is designed to explain the core functionality of the filter components, including information for those developers implementing custom filters.

Hierarchy

TCustomControl (VCL)

- TXpFilterBase (XpFltBas)

Properties

DisplayName	HPixelsPerInch	VPixelsPerInch

Methods

Reset	XslProcessorComment	XslProcessorProcessingInstr...
SaveToFile	XslProcessorElementEnd	XslProcessorQueryFilter
XslProcessorApplyStyleEnd	XslProcessorElementStart	XslProcessorResolveElement
XslProcessorApplyStyleStart	XslProcessorFoProperty	XslProcessorResolveElement...
XslProcessorAttribute	XslProcessorFormatObjectEnd	XslProcessorText
XslProcessorCDATASection	XslProcessorFormatObject...	

DisplayName — run-time, read-only property

```
property DisplayName : string
```

Specifies the name of the filter.

DisplayName provides a common name for a filter. For instance, when choosing a filter at run time DisplayName can be retrieved from the current filter and displayed to the user. The following table lists the DisplayNames for each of the supplied filters:

Filter	DisplayName
HTML	HTML
Print	PRINT
RTF	RTF
XML	XML

HPixelsPerInch — run-time property

```
property HPixelsPerInch : Integer
```

Default: 96

Specifies the number of pixels per inch horizontally.

Use this property to set or retrieve the number of horizontal pixels per inch. The provided HTML and RTF Filters use the default of 96 pixels per inch. The Print filter retrieves this setting by querying the setting from the current printer. Since the XML filter is not responsible for presentation, it does not use this property.

See also: VPixelsPerInch

Reset — virtual method

```
procedure Reset; virtual;
```

Clears the filter's result tree.

Each filter calls this method every time the XSL processor raises an OnApplyStyleStart event, so users shouldn't have to call this method directly.

See also: XslProcessorApplyStyleStart

SaveToFile — virtual method

```
function SaveToFile(sFile : string) : Boolean; virtual;
```

Saves the transformed and formatted document to disk.

Call this method the save the document produced by the HTML, RTF, and XML Filters to disk. Since the Print Filter's output is designed for the printer, it does not implement this method.

sFile is the location to save the file. If the save is successful, the function returns True. Otherwise, it returns False. The following example shows how to use this method:

```
procedure TfrmMain.btnSaveClick(Sender : TObject);
begin
  if (XpFilterHTML.SaveToFile(edtFileName.Text)) then
    memo1.Lines.Add(format('Weekly sales report saved to %s',
                           [edtFileName]))
  else
    memo1.Lines.Add('Weekly sales report could not be saved.');
end;
```

VPixelsPerInch — run-time property

```
property VPixelsPerInch : Integer
```

Default: 96

Specifies the number of vertical pixels per inch on the output document.

Use this property to set or retrieve the number of vertical pixels per inch. The provided HTML and RTF Filters use the default of 96 pixels per inch, but the Print filter retrieves its setting by querying the current printer. Since the XML filter is not responsible for presentation, it does not use this property.

See also: HPixelsPerInch

XslProcessorApplyStyleEnd — virtual method

```
procedure XslProcessorApplyStyleEnd(Sender : TObject); virtual;
```

Defines an event handler that is called after the XSL processor has applied the stylesheet to the XML document.

This is an event handler for the XSL processor's OnApplyStyleEnd event. It is the last event raised when the XSL processor's ApplyStyle method is called. It is used as notification that processing has completed.

If an error occurs while processing the stylesheet and the processor's RaiseErrors property is True, this handler won't be called. If there are errors and RaiseErrors is False, this handler will be called.

Sender is the XSL processor that raised the event.

See also: XslProcessorApplyStyleStart

XslProcessorApplyStyleStart — virtual method

```
procedure XslProcessorApplyStyleStart(Sender : TObject); virtual;
```

Defines an event handler that is called before the XSL processor has applied the stylesheet to the XML document.

This is an event handler for the XSL processor's OnApplyStyleStart event. The event simply notifies the handler that processing has begun. It is called after the XslProcessorQueryFilter. Sender is the XSL processor that raised the event.

See also: XslProcessorApplyStyleEnd, XslProcessorQueryFilter

XslProcessorAttribute — virtual method

```
procedure XslProcessorAttribute(
  oOwner : TObject; sName, sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called when the XSL processor sends an attribute to the result tree.

This is an event handler for the XSL processor's OnAttribute event. The XSL processor's OnAttribute reference explains when this handler is called. oOwner is the XSL processor raising the event. sName is the name of the attribute and sValue is the value of the attribute.

XslProcessorCDATASection — virtual method

```
procedure XslProcessorCDATASection(oOwner : TObject;
  sValue : DOMString; bReplace : Boolean); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called when the XSL processor sends a CDATA section to the result tree.

This is an event handler for the XSL processor's OnCDATASection event. oOwner is the XSL processor raising the event. sValue is the data contained in the CDATA section. If bReplace is True, filter implementations of SGML and its subsets (HTML, XML, etc.) should replace special characters with their entity representations. The following code is an example handler for this event:

```
procedure TCustomFilter.XslProcessorCDATASection(
  oOwner : TObject; sValue : DOMstring; bReplace : Boolean);
var
  oText : TXpCDATASection;
begin
  if FCurrent <> nil then begin
    { Replace special chars with entities in sValue }
    if bReplace then begin
      sValue := StringReplace(
        sValue, '&', '&amp;', [rfReplaceAll]);
      sValue := StringReplace(
        sValue, '<', '&lt;', [rfReplaceAll]);
    end;

    oText := FQueueCurrent.Document.CreateCDATASection(sValue);
    FCurrent.AppendChild(oText);
    oText.Release;
  end;
end;
```

XslProcessorComment — virtual method

```
procedure XslProcessorComment(
  oOwner : TObject; sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called when a comment is sent to the result tree.

This is an event handler for the XSL processor's OnComment event. Comments are defined in an XSL stylesheet using the xsl:comment command. oOwner is the XSL processor raising the event. sValue is the text of the comment.

XslProcessorElementEnd — virtual method

```
procedure XslProcessorElementEnd(
  oOwner : TObject; sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called after the XSL processor has passed an element to the result tree.

This is an event handler for the XSL processor's OnElementEnd event. See the XSL processor's OnElementEnd and OnElementStart references for additional information.

oOwner is the XSL processor raising the event. oOwner is the XSL processor raising the event. sValue is the name of the element.

See also: XslProcessorElementStart

XslProcessorElementStart — virtual method


```
procedure XslProcessorElementStart(
  oOwner : TObject; sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called before an XSL processor sends an element to the result tree.

This is the event handler for the XSL processor's OnElementStart event. See the XSL processor's OnElementEnd and OnElementStart references for additional information.

oOwner is the XSL processor event that raised the event. sValue is the name of the element.

See also: XslProcessorElementEnd

XslProcessorFoProperty — virtual method

```
procedure XslProcessorFoProperty(oOwner : TObject;
  wCode : Integer; sName, sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called for each property of a formatting object in the stylesheet.

This is the event handler for the XSL processor's OnFoProperty event. Formatting object properties are the details pertaining to formatting objects. The properties can include margin width, background color, and font type. The properties appear as attributes of formatting object elements in an XSL stylesheet. Formatting objects and their related properties and explained in Chapter 9.

oOwner is the XSL processor raising the event. wCode is the formatting object property tag code as shown in the Appendix. sName is the name of the property and sValue contains the property's value.

See also: XslProcessorFormatObjectEnd, XslProcessorFormatObjectStart

XslProcessorFormatObjectEnd — method

```
procedure XslProcessorFormatObjectEnd(
  Sender : TObject; aFOType : Integer); virtual;
```

Defines an event handler that is called after a formatting object is processed.

This is the event handler for the XSL processor's OnFormatObjectEnd event. Sender is the XSL processor raising the event. aFOType is the formatting object tag code as listed the Appendix.

See also: XslProcessorFormatObjectStart

XslProcessorFormatObjectStart — method

```
procedure XslProcessorFormatObjectStart(Sender : TObject;
  aFOType : Integer; const aData : array of const); virtual;
```

Defines an event handler that is called before the XSL processor processes a formatting object.

This is the event handler for the XSL processor's OnFormatObjectStart event. Sender is the XSL processor raising the event. aFOType is the formatting object tag code as shown in the Appendix. aData contains the parameters required for the formatting object.

See also: XslProcessorFormatObjectEnd

XslProcessorProcessingInstruction — virtual method

```
procedure XslProcessorProcessingInstruction(
  oOwner : TObject; sName, sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called when the XSL processor passes a processing instruction to the result tree.

This is the event handler for the XSL processor's OnProcessingInstruction event. oOwner is the XSL processor raising the event. sName is the name of the processing instruction and sValue is its value.

XslProcessorQueryFilter — virtual method

```
procedure XslProcessorQueryFilter(
  Sender : TObject; var sName : string); virtual;
```

Defines an event handler that is called when the XSL processor queries for the name of the attached filter.

This is the event handler for the XSL processor's OnQueryFilter event. This is the first handler called after the XSL processor's ApplyStyle is called. Sender is the XSL processor raising the event.

Each custom filter should assign a human-friendly display name to the protected FDisplayName class member and assign its value to sName in this handler. The following method is an example of this handler:

```
procedure TCustomFilter.XslProcessorQueryFilter(
  Sender : TObject; var sName : string);
begin
  sName := FDisplayName;
end;
```


See also: XslProcessorApplyStyleStart

XslProcessorResolveElement — virtual method

```
procedure XslProcessorResolveElement(oOwner : TObject;
  wCode : Integer; var sTag : string); virtual;
```

Defines an event handler that is called when the XSL processor tries resolve an XSL tag code to an element name.

This is the event handler for the XSL processor's OnResolveElement event. oOwner is the XSL processor raising the event. wCode is the XSL tag code of the element. sTag is the stylesheet command. The XSL tag codes are listed in the Appendix.

Note: See the OnResolveElement event of the TXpXSLProcessor component and the OnResolveElementCode event of the TXpXSLProcessor for additional information.

See also: XslProcessorResolveElementCode

XslProcessorResolveElementCode — virtual method

```
procedure XslProcessorResolveElementCode(oOwner : TObject;
  sElement : string; var wCode : Integer); virtual;
```

Defines an event handler that is called when the XSL processor tries to resolve an element's name to an XSL tag code.

This is the event handler for the XSL processor's OnResolveElementCode event. This event is called for each element sent to the result tree. oOwner is the XSL processor raising the event. sElement is the XSL stylesheet command, if the element is a stylesheet command. If the element's name isn't a stylesheet command, sElement is an empty string. wCode is the element's XSL tag code. XSL tag codes are listed in the Appendix.

Note: See the XSL processor references for OnResolveElement and OnResolveElementCode for additional information.

See also: XslProcessorResolveElement

XslProcessorText — virtual method

```
procedure XslProcessorText(
  oOwner : TObject; sValue : DOMString); virtual;
```

```
DOMString = WideString;
```

Defines an event handler that is called when the XSL processor sends text to the result tree.

This is the event handler for the XSL processor's OnText event. See the XSL processor's OnText event reference to see when this event is raised. oOwner is the XSL processor raising the event. sValue is the text being added to the result tree.

TXpFilterHTML Component

The HTML filter component, TXpFilterHTML works with the XSL processor to produce HTML documents. As described in the rest of this section, the component has options to produce HTML as virtually any application should require.

Hierarchy

TCustomControl (VCL)

- ❶ TXpFilterBase (XpFltBas) 263
  - TXpFilterHTML (XpFltHtm)

Properties

- CSS
- ❶ DisplayName
- FormattedOutput
- ❶ HPixelsPerInch
- HTMLDocument
- OutputEncoding
- OutputMode
- ❶ VPixelsPerInch
- XMLValidHTML

Methods

- ❶ Reset
- ❶ SaveToFile
- ❶ XslProcessorApplyStyleEnd
- ❶ XslProcessorApplyStyleStart
- ❶ XslProcessorAttribute
- ❶ XslProcessorCDATASection
- ❶ XslProcessorComment
- ❶ XslProcessorElementEnd
- ❶ XslProcessorElementStart
- ❶ XslProcessorFoProperty
- ❶ XslProcessorFormatObjectEnd
- ❶ XslProcessorFormatObject...
- ❶ XslProcessorProcessingInstr...
- ❶ XslProcessorQueryFilter
- ❶ XslProcessorResolveElement
- ❶ XslProcessorResolveElement...
- ❶ XslProcessorText

References

CSS — property

```
property CSS : Boolean
```

Default: True

Specifies whether cascading stylesheet (CSS) data is created within the HTML document.

When CSS is True, CSS data is include within the HTML document. When it is False, CSS data is not included with the HTML document.

This is an example HTML document when this value is False:

```
<BODY leftmargin="36.0pt" topmargin="36.0pt" rightmargin="36.0pt"
bottommargin="36.0pt"><SPAN font-weight="bold">To: Paul
Prescot<br>From: Jack Duncan<br>Date: Mon, 21 Apr 1997 09:27:55
+0200<br>Subject: XML literature<br><br>Message:<br>Hello Mr
<u>Behme</u>,Please read <u>Jon Bosak</u>'s introductory
text"SGML, Java and the Future of the Web"Best wishes,
<u>Jack</u></SPAN>
</BODY>
```

This is the same HTML document with CSS set to True:

```
<HTML><HEAD><TITLE></TITLE></HEAD><BODY leftmargin="36.0pt"
topmargin="36.0pt" rightmargin="36.0pt"
bottommargin="36.0pt"><SPAN STYLE="d; d; d; d; font-weight:
bold">To: Paul Prescot<br>From: Jack Duncan<br>Date: Mon, 21 Apr
1997 09:27:55 +0200<br>Subject: XML
literature<br><br>Message:<br>Hello Mr <u>Behme</u>,Please read
<u>Jon Bosak</u>'s introductory text"SGML, Java and the Future of
the Web"Best wishes, <u>Jack</u></SPAN></BODY></HTML>
```

FormattedOutput — property

```
property FormattedOutput : Boolean
```

Default: True

Indicates whether the HTML document has line breaks to improve readability.

When this value is False, the filter produces HTML as a single line without any line breaks. Setting this property to True causes the filter to insert line breaks after each HTML element which makes it more human readable, but does produce a larger HTML document.

See also: OutputMode

HTMLDocument — run-time, read-only property

```
property HTMLDocument : DOMString
```

Retrieves the HTML document.

After the XSL stylesheet has been applied to the XML document, retrieve the resulting HTML document by getting the value of this property.

See also: SaveToFile

OutputEncoding — property

```
OutputEncoding : TXpCharEncoding
```

```
TXpCharEncoding = (
  ceUnknown, ceUTF8, ceUTF16LE, ceUTF16BE, ceISO88591);
```

Default: ceUTF8

Determines the character encoding of the HTML document.

This property determines the character encoding of the retrieved HTML document. Use the following table to determine the proper setting for this property:

Value	Output Format
ceUnknown	This is used internally. Do not assign this value to the property.
ceUTF8	UTF-8 character encoding. Each character is represented using between 1 and 6 bytes.
ceUTF16LE	UTF-16 character encoding with little endian byte order. Each character is represented with two bytes.
ceUTF16BE	UTF-16 character encoding with big endian byte order. Each character is represented using two bytes.
ceISO88591	ISO-8859-1 character encoding. Also known as Latin-1. This is a subset of UTF-8 that represents each character in a single byte.

See also: HTMLDocument, SaveToFile

OutputMode property

```
property OutputMode : TXpHTMLOutputMode
```

```
TXpHTMLOutputMode = (XphomBodyOnly, XphomFragment, XphomFull);
```

Default: XphomFull

Specifies the content of the HTML document.

This setting determines how the document is retrieved when using the HTMLDocument property or calling the SaveToFile method. The following table explains the options for this property. If using a stylesheet that outputs all HTML markup required in the output document, then set this property to xphomFragment.

Value	Meaning
XphomBodyOnly	Only the body region of the result tree is retrieved. The HTML document includes HTML, HEAD, and BODY tags. The before, after, start, and end regions of the result tree are not converted to HTML.
XphomFragment	Same as XphomBodyOnly, but does not include the stylesheet (CSS) formatting. The HTML, HEAD, and BODY tags are also not retrieved.
XphomFull	Produces the complete HTML document with stylesheet formatting and HTML, HEAD, and BODY tags.

See also: CSS, FormattedOutput, OutputEncoding

XMLValidHTML property

```
property XMLValidHTML : Boolean
```

Default: True

Specifies whether HTML document should include end tags.

When XMLValidHTML is set to True, all end tags are included in the HTML document. Optional end tags are omitted when set to False.

TXpFilterPrint Component

The Print filter is a visual component that contains a document that can be previewed and printed. Since this is a visual control it makes the following TCustomControl properties and events visible:

Align	BevelWidth	OnMouseMove
BevelEdges	BorderWidth	OnMouseUp
BevelInner	OnClick	OnResize
BevelKind	OnDblClick	
BevelOuter	OnMouseDown	

See TCustomControl in your IDE's online help for information about these items.

Hierarchy

7

TCustomComponent (VCL)

❶ TXpFilterBase (XpFltBas) 263

TXpFilterPrint (XpFltPrt)

Properties

❶ DisplayName	HSlidePosition	❶ VPixelsPerInch
❶ HPixelsPerInch	PageCount	VSlideFactor
HSlideFactor	PageCurrent	VSlidePosition

Methods

Print	❶ XslProcessorCDATASection	❶ XslProcessorFormatObject...
❶ Reset	❶ XslProcessorComment	❶ XslProcessorProcessingInstr...
❶ SaveToFile	❶ XslProcessorElementEnd	❶ XslProcessorQueryFilter
❶ XslProcessorApplyStyleEnd	❶ XslProcessorElementStart	❶ XslProcessorResolveElement
❶ XslProcessorApplyStyleStart	❶ XslProcessorFoProperty	❶ XslProcessorResolveElement...
❶ XslProcessorAttribute	❶ XslProcessorFormatObjectEnd	❶ XslProcessorText

HSlideFactor — read-only, run-time property

```
property HSlideFactor : Integer
```

Returns the number of pixels that the control will scroll horizontally.

This is useful for implementing scroll bars outside of the component to control the horizontal viewing of the document.

See also: HSlidePosition, VSlideFactor, VSlidePosition

HSlidePosition — property

```
property HSlidePosition : Integer
```

Controls the horizontal position of the rendered page inside the component window.

Valid values are from 0 to HSlideFactor.

See also: HSlideFactor, VSlideFactor, VSlidePosition

PageCount — read-only, run-time property

```
property PageCount : Integer
```

Returns the number of pages in the current document.

This value of this property available after ApplyStyle completes.

See also: PageCurrent

PageCurrent — property

```
property PageCurrent : Integer
```

Determines the current page in the display control.

Valid values are 1 to PageCount. Setting this property causes the component to refresh its display.

See also: PageCount

```
procedure Print(sPages : string);
```

Prints selected pages of the rendered document.

The document is assigned to the currently assigned printer. At the time that ApplyStyle is called, the component gathers information from the current printer's paper selection and orientation for determining the look of the printed document. Document pagination is based on this information. After ApplyStyle is completed, the component displays the first page of the rendered document.

Note: This component does not provide any means of setting the current printer or its settings. This functionality can easily be added to a VCL application using a TPrintDialog component. At the time of this writing, this functionality is still being finalized for CLX applications.

Note: If the current printer or printer settings change after calling ApplyStyle, the current representation may not reflect current printer settings and may produce unpredictable results. This can be remedied by calling ApplyStyle again.

sPages represents the pages to print. Passing an empty string prints all pages of the document. Print individual pages by separating each page number with a comma. Print a range of pages by placing a dash between the beginning and ending page numbers. Following are some examples:

Print all pages:

```
PrtFilter.Print('');
```

Print selected pages:

```
PrtFilter.Print('1,3,5,7');
```

Print a range of pages:

```
PrtFilter.Print('1-9');
```

Print selected pages and a range of pages:

```
PrtFilter.Print('1-5,10,20,30');
```

See also: PageCount

VSlideFactor — run-time, read-only property

```
property VSlideFactor : Integer
```

Returns the number of pixels that the control will scroll vertically.

This is useful for implementing scroll bars outside of the component to control the vertical viewing of the document.

See also: HSlideFactor, HSlidePosition, VSlidePosition

VSlidePosition — property

```
property VSlidePosition : Integer
```

Controls the vertical position of the rendered page inside the component window.

Valid values are from 0 to VSlideFactor.

See also: HSlideFactor, HSlidePosition, VSlideFactor

TXpFilterRTF Component

The RTF filter is designed to produce documents according to Microsoft's Rich Text Format (RTF) specification. RTF is a file specification that allows embedding formatting information within the text. One place that RTF is used is the TRichEdit control. The TRichEdit control uses RTF to provide bold, underlined, and italicized text.

Hierarchy

TCustomControl (VCL)

- ❶ TXpFilterBase (XpFltBas) 263
  - TXpFilterRTF (XpFltRTF)

Properties

- ❶ DisplayName
- ❶ HPixelsPerInch
- RTFDocument
- ❶ VPixelsPerInch

Methods

- ❶ Reset
- ❶ SaveToFile
- ❶ XslProcessorApplyStyleEnd
- ❶ XslProcessorApplyStyleStart
- ❶ XslProcessorAttribute
- ❶ XslProcessorCDATASection
- ❶ XslProcessorComment
- ❶ XslProcessorElementEnd
- ❶ XslProcessorElementStart
- ❶ XslProcessorFoProperty
- ❶ XslProcessorFormatObjectEnd
- ❶ XslProcessorFormatObject...
- ❶ XslProcessorProcessingInstr...
- ❶ XslProcessorQueryFilter
- ❶ XslProcessorResolveElement
- ❶ XslProcessorResolveElement...
- ❶ XslProcessorText

RTFDocument — read-only, run-time property

```
property RTFDocument : string
```

Retrieves the rendered RTF document.

After the XSL processor's ApplyStyle has successfully completed, the rendered RTF document can be retrieved using this property.

See also: SaveToFile

TXpFilterXML Component

The XML filter produces and XML document.

Hierarchy

TCustomComponent (VCL)

- ❶ TXpFilterBase (XpFltBas) 263
  - TXpFilterXML (XpFltXML)

Properties

- BodyOnly
- ❶ DisplayName
- FormattedOutput
- ❶ HPixelsPerInch
- OutputEncoding
- ❶ VPixelsPerInch
- XMLDocument

Methods

- ❶ Reset
- ❶ SaveToFile
- ❶ XslProcessorApplyStyleEnd
- ❶ XslProcessorApplyStyleStart
- ❶ XslProcessorAttribute
- ❶ XslProcessorCDATASection
- ❶ XslProcessorComment
- ❶ XslProcessorElementEnd
- ❶ XslProcessorElementStart
- ❶ XslProcessorFoProperty
- ❶ XslProcessorFormatObjectEnd
- ❶ XslProcessorFormatObject...
- ❶ XslProcessorProcessingInstr...
- ❶ XslProcessorQueryFilter
- ❶ XslProcessorResolveElement
- ❶ XslProcessorResolveElement...
- ❶ XslProcessorText

Reference Section

BodyOnly — property

```
property BodyOnly : Boolean
```

Default: False

Determines if only the body region of the XML document should be retrieved.

When this property is False, all regions of the result tree are retrieved by the SaveToFile method and XMLDocument property. When this property is True, only the body region is retrieved by SaveToFile and XMLDocument.

See also: SaveToFile, XMLDocument

FormattedOutput — property

```
property FormattedOutput : Boolean
```

Default: True

Indicates whether the XML document has line breaks to improve readability.

When this value is False, the filter produces XML as a single line without any line breaks. Setting this property to True causes the filter to insert line breaks after each XML element and indent lines according to the element hierarchy. This makes it more human readable, but does produce a larger document.

OutputEncoding — property

```
property OutputEncoding : TXpCharEncoding
```

```
TXpCharEncoding = (
  ceUnknown, ceUTF8, ceUTF16LE, ceUTF16BE, ceISO88591);
```

Default: ceUTF8

Determines the character encoding of the HTML document.

Since not all characters can be properly represented using a single byte, this property allows XML documents to be retrieved with a variety of Unicode character coding formats. Use the following table when setting this property:

Value	Output Format
ceUnknown	This setting is used internally and should not be assigned to this property.
ceUTF8	UTF-8 character encoding. Each character is represented using between 1 and 6 bytes.
ceUTF16LE	UTF-16 character encoding with little endian byte order. Each character is represented with two bytes.
ceUTF16BE	UTF-16 character encoding with big endian byte order. Each character is represented using two bytes.
ceISO88591	ISO-8859-1 character encoding. Also known as Latin-1. This is a subset of UTF-8 that represents each character in a single byte.

XMLDocument — read-only, run-time property

```
property XMLDocument : DOMString
```

Retrieves the HTML document.

After the XSL processor's ApplyStyle has successfully completed, the rendered XML document can be retrieved using this property.

See also: BodyOnly, SaveToFile

Chapter 8: XSLT Reference

A stylesheet is implemented using elements from the XSL namespace. The first part of this chapter discusses the concepts and tools used within the XSLT elements, such as XPath expressions, patterns, and attribute value templates. The remainder of the chapter serves as a reference, describing each XSLT element in detail. Please note that the information provided in this chapter covers the everyday aspects of each element. However, it does not cover all of the nuances found in the original XSLT 1.0 specification or in a commercial book dedicated to XSLT.

XPath Expressions

XPath, like the Document Object Model (DOM), models a XML document as a tree of nodes. An XPath *expression* is a mechanism for navigating through and selecting nodes from the XML document. An expression is analogous to a Structured Query Language (SQL) query used to select records from a database.

An expression may return one of the result types described in Table 8.1.

Table 8.1: *XPath expression result types*

Result Type	Description
Boolean	The value True or False.
Node-set	An unordered collection of nodes. There are no duplicates within the collection.
Number	A floating point number.
String	A sequence of characters.

8

An expression may contain string or numeric constants. String constants may be surrounded with single or double quotes (e.g., 'book', "publisher"). Since expressions often occur as an attribute value, they are most often delimited using single quotes. Numeric constants may be integers or floating point numbers (e.g., 2, 4.55, -8.01). Boolean constants must be declared using the true() and false() functions.

An expression may contain a reference to a local or global variable. The variable name must be prefixed with a dollar character ($). A variable may be referenced only when it is in scope and the variable may contain any type of value (e.g., boolean, node-set, number, string).

An expression may also contain a function reference. The processor maps the function name to the corresponding XPath or XSLT function. For more information concerning XPath and XSLT functions, see the XPath 1.0 and XSLT 1.0 specifications.

Location paths and steps

One of the most important kinds of expressions is the location path. A location path selects a set of nodes relative to the current node. A location path always returns a result type of node-set. Location paths may be relative or absolute. A relative location path starts at the current node and selects a set of nodes using one or more location steps. For example, the relative location path "Book" selects all Book nodes that are child elements of the current node. An absolute location path starts at the document root (denoted by the character '/') and selects a set of nodes using one or more location steps. For example, the absolute location path "/Book" selects the Book node that is a child element of the document root.

A location step has three parts:

- An axis, which specifies the direction of travel starting from the current node.
- A node test, which specifies the node type and expanded name of the nodes selected by the location step.
- Zero or more predicates, which use expressions to filter nodes selected by the location step.

The format of a location step is as follows:

```
axis-name::node-test[predicate 1]..[predicate N]
```

Remember that predicates are optional. A location path may contain one or more location steps where each location step is separated by the forward slash character ('/').

Axis names

Table 8.2 lists the axes that may be specified within a location step.

Table 8.2: *Axis names and descriptions*

Axis Name	Abbreviated Name	Node-Set Members
ancestor	N/A	The ancestors of the current node. An ancestor is the parent of the current node, the parent's parent, and so on.
ancestor-or-self	N/A	The current node and its ancestors. An ancestor is the parent of the current node, the parent's parent, and so on.
attribute	@	The attributes of the current node. If the current node is not an element then this axis is empty.
child	If no axis name is specified, the processor defaults to the child axis.	The children of the current node.
descendant	Double forward slash ("//")	The descendants of the current node, where a descendant is a child node, the child of the child, and so on.

Table 8.2: *Axis names and descriptions (continued)*

Axis Name	Abbreviated Name	Node-Set Members
descendant-or-self	N/A	The current node and its descendants. A descendant is a child node, the child of the child, and so on.
following	N/A	All nodes that are after the current node, in document order, excluding an descendants, attribute nodes, and namespace nodes.
following-sibling	N/A	All of the following siblings of the current node. If the current node is an attribute or namespace node then this axis is empty.
namespace	N/A	The namespace nodes of the current node. If the current node is not an element then this axis is empty.
parent	Double period (“..”)	The parent of the current node, if there is one.
preceding	N/A	All nodes that precede the current node, in document order, excluding any ancestors, attribute nodes, and namespace nodes.
preceding-sibling	N/A	All of the siblings preceding the current node. If the current node is an attribute or namespace node then this axis is empty.
self	Period character (‘.’)	The current node.

Node tests

The node test within a location step may test a node’s name or type. If the node fails the test then it is not included in the node-set. A name test may involve a literal name or wildcard pattern. For example, the node test “frog” matches only those nodes of having the name “frog”. The node test ‘*’ matches all nodes regardless of name. The node test

“tpx:newsgroup” matches nodes having the name “newsgroup” within the “tpx” namespace. Namespace comparisons are performed using the namespace URI, not the namespace prefix.

A node type test involves the type of node that may be found on the current axis and a node type function. The attribute axis contains attribute nodes only. The namespace axis contains namespace nodes only. All other axes contain element nodes. Table 8.3 identifies the node type functions that may be used in an node test.

Table 8.3: *Node type functions*

Function	Tests For
`comment()`	Comment nodes
`node()`	Any type of node
`processing-instruction()`	Processing instruction nodes
`text()`	Text nodes

When using the attribute axis, a node test involving the comment() function would return an empty node set. That is because the attribute axis contains attribute nodes and the comment() function looks only for comment nodes.

8

Predicates

A predicate refines the set of nodes built using an axis and node test. A location step may contain zero or more predicates. The predicates are applied, in the order they are listed, to each node within the node-set. During the evaluation, the node being test becomes the current node. If the node fails to meet the requirement stated by any predicate then it is removed from the node-set.

Each predicate is surrounded by left and right brackets (e.g., ‘[’ and ‘]’). A predicate may test for position (e.g., is this the first child node of the current node) or contain an expression that is evaluated for the current node. For example, the location step “animal[1]” looks for the first child animal node of the current node.

Examples

Table 8.4 contains a number of example location paths, using the information outlined in the previous sections.

Table 8.4: *Example paths*

Location Path	Abbreviated Path	Purpose
`child::animal`	`animal`	Selects the animal child elements of the current node.
`child::*`	`*`	Selects all child elements of the current node.
`child::text()`	`text()`	Selects all text node children of the current node.
`child::text()[position()=2]`	`text()[2]`	Selects the second text node child of the current node.
`attribute::name`	`@name`	Selects the name attribute of the current node.
`attribute::*`	`@*`	Selects all attribute nodes of the current node.
`descendant::keeper`	`.//keeper`	Selects all keeper element descendants of the current node.

Table 8.4: *Example paths*

Location Path	Abbreviated Path	Purpose
child::zoo/child::city	zoo/city	Selects the city grandchildren of the zoo child elements of the current node.
/descendant::zoo	//zoo	Selects all zoo elements in the same document as the current node
child::zoo[position()=last()]	zoo[last()]	Selects the last zoo child element of the current node.
parent::*/attribute::name	../@name	Selects the name attribute of the parent element of the current node.

8

Table 8.4: *Example paths*

Location Path	Abbreviated Path	Purpose
descendant::animal [attribute::type="fish"]	.//animal[@type="fish"]	Selects all descendant animal elements having a type attribute having the value "fish".
descendant::animal [attribute::type="fish"] [position()=1]	.//animal[@type="fish"][1]	Selects the first descendant animal element having a type attribute having the value "fish".
child::animal [attribute::type and attribute::genus]	animal[@type and @genus]	Selects those animal child elements having both a type and genus attribute.

Patterns

A pattern is used to verify that a node matches one or more conditions. A node that satisfies all of the conditions matches the pattern. Patterns are used primarily to determine which templates match the current node being handled by the XSL processor (i.e., the xsl:template element). However, patterns are also used by the xsl:key and xsl:number elements.

A pattern may be formed using a subset of the XPath expression syntax. In general, any expression that contains an absolute or relative location path operating on the child or attribute axis is a valid pattern. Table 8.5 lists examples of valid patterns.

Table 8.5: *Valid patterns*

Pattern	Matches
`invoice`	Any invoice element
`*`	Any element
`invoice|creditmemo`	Any invoice or creditmemo element
`invoice/lineitem`	Any lineitem element having an invoice element as its parent
`distribcenter//location`	Any location element having a distribcenter element as an ancestor
`node()`	Any node other than an attribute node and the document root
`/`	The document root
`*[1 and self::uom]`	Any uom element that is the first child element of its parent
`@price`	Any price attribute
`@*`	Any attribute

Attribute Value Templates

Attributes within literal result elements and within certain XSLT elements may be evaluated as *attribute value templates*. An attribute value template contains a XPath expression surrounded by curly braces ('{' and '}'). When an attribute value template is encountered, the XSL processor evaluates the expression and converts it to a string. Use an attribute value template to compute attribute values at run time.

For example, the following literal result element is an HTML table row that computes its background color:

```
<TR bgcolor="{@color}">
  ...
</TR>
```

Note that an attribute of an XSLT element is evaluated as an attribute value template only if the XSLT specification explicitly states the attribute is treated as an attribute value template.

xsl:apply-imports

```
<xsl:apply-imports/>
```

Invokes an overridden template rule.

This element is defined in Section 5.6 of the XSLT 1.0 specification.

In XSLT, it is possible to override a template rule in an imported stylesheet with another template rule. Use the xsl:apply-imports element to invoke the overridden template rule. For example, stylesheet A.xsl contains a template rule for example elements:

```
<xsl:template match="example">
  <pre><xsl:apply-templates/></pre>
>/xsl:template>
```

Stylesheet B.xsl imports A.xsl and overrides the template rule for example elements:

```
<xsl:import href="A.xsl"/>
<xsl:template match="example">
  <div style="border: solid red">
    <xsl:apply-imports/?
  </div>
</xsl:template>
```

The result tree would contain the following for each example element:

```
<div style="border: solid red"><pre>…</pre></div>
```

For another example, see the Importing Stylesheets section on page 227.

Note that xsl:apply-imports uses the concept of a *current template rule*. When the XSL processor invokes a template, that template becomes the current template rule during the instantiation of the template's body. The xsl:apply-imports element searches for a template rule matching the current template rule. The search returns the template rule with the next lowest import precedence. If the current template rule has a mode, the search returns a template rule with the next lowest import precedence having the same mode as the current template.

If there is no current template rule then xsl:apply-imports raises an exception. One situation in which there is no current template rule is during the instantiation of an xsl:for-each element.

See also: xsl:import

```
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!--Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

Applies template rules to a set of nodes.

This element is defined in Section 5.4 of the XSLT 1.0 specification.

Use this element to have the XSL processor search for and apply template rules to a set of nodes. The xsl:apply-templates element is always used within a template body. One or more xsl:apply-templates elements may occur within the same template body.

Use the select attribute to specify a set of nodes. The select attribute is optional. If it exists, it must take the form of an XPath expression that returns a node set. An XPath expression returning some other type of value such as a number or string will cause an exception to be raised. The node set may be relative to the current node or relative to the document root. The expression may also reference a variable declared via the xsl:variable element. For information concerning XPath expressions, see “XPath Expressions” on page 286 .

The following example applies templates to all book elements having a certain publisher:

```
<xsl:template match="books">
  ...
  <xsl:apply-templates select="book[@publisher=$pub]"/>
</xsl:template>
```

If no select attribute is specified, the XSL processor applies templates to the child nodes of the current node. The following example creates a block for a chapter element and then processes its immediate children:

```
<xsl:template match="chapter">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Note that attributes are not considered to be child nodes. To access attributes, you must specify them via the select attribute as shown in the following example:

```
<xsl:apply-templates select="@name" />
```

The mode attribute is optional. If the mode attribute is not present, the XSL processor only considers those templates not having a mode attribute. If the mode attribute is specified, the XSL processor uses only those templates with a matching mode. For example, the following stylesheet generates an HTML document containing a table of contents followed by the text for each topic:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <xsl:apply-templates select="//topic" mode="toc"/>
  <xsl:apply-templates select="//topic"/>
</xsl:template>

<xsl:template match="topic" mode="toc">
  <A href="#{.}">
  <xsl:value-of select="."/>
  </A>
  <br/>
</xsl:template>

<xsl:template match="topic">
  <H3>
    <A name="{.}">
    <xsl:value-of select="."/>
    </A>
  </H3>
  <br/>
</xsl:template>
</xsl:stylesheet>
```

The template for the document root first applies template rules for all topics elements with mode “toc.” It then applies template rules for all topics without a mode. The template rule for topics with mode “toc” creates a hyperlink to a bookmark created in the template rule for topics without a mode.

The xsl:apply-templates body may contain zero or more xsl:sort elements along with zero or more xsl:with-param elements. If there are no xsl:sort elements, the nodes are selected in the order in which they are encountered in the document. If one or more xsl:sort elements are present, the node set is sorted before the template rules are applied. For more information about sorting, see the xsl:sort element on page 339. The xsl:sort elements may be intermixed with the xsl:with-param elements.

Use the xsl:with-param elements to specify parameters that are made visible to the invoked templates. Each template rule invoked by xsl:apply-templates has access to each parameter. XMLPartner evaluates each xsl:with-parameter element only once, before template rules are applied to the node set. In the following example, the stylesheet applies a 10% discount to the orders for a particular customer:

```
<xsl:apply-templates select="order[@customerID=$custID]">
  <xsl:with-param name="discountPct" select="10"/>
</xsl:apply-templates>
```

For a discussion of how the XSL processor chooses template rules when one or more template rules match a node, see the description of the xsl:template element or “Handling template collisions” on page 228.

See also: xsl:sort, xsl:template, xsl:variable, xsl:with-param

xsl:attribute

```
<xsl:attribute
  name = qname
  namespace = uri-reference >
  <!—Content: template -->
</xsl:attribute>
```

8

Creates an attribute in current element of the result tree.

This element is defined in Section 7.1.3 of the XSLT 1.0 specification.

Use this element to add an attribute name and value to the current element node of the result tree. The attribute must be added before any child elements are added to the current element. If the current element already has child elements then xsl:attribute raises an exception. The xsl:attribute element may be used in a template body or within the xsl:attribute-set element.

The template body of the xsl:attribute element produces the value of the new attribute. The name attribute is required and may take the form of an explicit name or an *attribute value template*. An attribute value template is an XPath expression nested within curly braces (i.e., ‘{’; and ‘}’). The XSL processor evaluates the expression and uses its value as the name of the attribute. The resulting name must be a valid Qualified Name (i.e., QName) having a local name and an optional namespace prefix (e.g., “style”, “ams:distribClass”).

The following example contains a template that uses the name attribute of a bookmark element to create a bookmark in an XML document:

```
<xsl:template match="bookmark">
  <A>
    <xsl:attribute name="name">
     <xsl:value-of select="@name">
    </xsl:attribute>
    <xsl:apply-templates>
  </A>
</xsl:template>
```

For a bookmark element having a name attribute with the value “WhatsNew” and containing a node with a template rule resulting in the text “What’s New in Release 2.51”, the resulting HTML is as follows:

```
<A name="WhatsNew">What's New in Release 2.51</A>
```

The namespace attribute is optional. If present, it may take the form of an explicit URI or an attribute value template as described earlier in this section. XMLPartner does not validate the URI so this attribute could have any string value.

If the name attribute does not contain a namespace prefix then the XSL processor will create a prefix for the attribute name. For example, the following template is evaluated:

```
<xsl:template match="bookmark">
  <A>
    <xsl:attribute name="name"
                   namespace="http://www.turbopower.com">
      <xsl:apply-templates>
    </xsl:attribute>
</xsl:template>
```

The resulting output would look like the following:

```
<A name="xpns1:WhatsNew"
  xmlns:xpns1="http://www.turbopower.com">
What's New in Release 2.51</A>
```

If another attribute of the element contains a matching namespace prefix but the namespace URI is different then the XSL processor will adjust the namespace prefix of the new attribute.

If the name attribute contains a namespace prefix but no namespace attribute is specified, the XSL processor searches for a namespace declaration matching the prefix. If one is found then that URI is used to create a namespace declaration for the attribute. Otherwise, no namespace declaration is created.

If the attribute value contains a new line character then the new line character is converted into a character reference. This is best shown with the following example:

```
<xsl:attribute name="a">x
y</xsl:attribute>
```

The example created the following attribute:

```
a="x&#13;y"
```

If the current element already contains an attribute with the same expanded name as the new attribute then the new attribute replaces the existing attribute.

See also: xsl:attribute-set, xsl:element

xsl:attribute-set

```
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

8

Defines a named set of attributes.

This element is defined in Section 7.1.4 of the XSLT 1.0 specification.

Use the xsl:attribute-set element to define a collection of attributes. The entire set of attributes may then be assigned to an element created via the xsl:element or xsl:copy elements. This is a quick way to define styles for an HTML table or a formatting object. As a top-level element, xsl:attribute-set may occur only as a child of an xsl:stylesheet element.

The name attribute of an xsl:attribute-set element may take the form of a qualified name (i.e., qname), having a local name and an optional namespace prefix. If the name includes a prefix then it must match a namespace declaration that is currently in scope.

The body of this element may contain zero or more xsl:attribute elements. When an xsl:element or xsl:copy element references an attribute set, the attributes are instantiated within the new element. The body of each xsl:attribute element is instantiated each time the attribute set is used.

A literal xsl:attribute-set element may include other attribute sets via its use-attribute-sets attribute. If specified, use-attribute-sets must contain zero or more white-space separated names of other attribute sets within the same stylesheet. The attributes from the referenced attribute sets are merged into the literal attribute set. The referenced attribute sets are merged in the same order as they occur in the use-attribute-sets attribute. Their attributes are inserted prior to the xsl:attribute elements of the literal attribute set.

Attributes specified in a referenced attribute set may be overridden by declaring the attribute in the literal attribute set. The XSL processor merges attribute sets having the same name. If the XSL processor encounters two or more attributes with the same qualified name, the XSL processor uses the attribute from the attribute set with the highest import precedence. For information on import precedence, see “Handling template collisions” on page 228.

If the attribute set directly or indirectly references itself then the XSL processor raises an exception.

The following stylesheet creates a named attribute set title-style and uses it in a template rule:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:attribute-set name="title-style">
  <xsl:attribute name="font-size">12pt</xsl:attribute>
  <xsl:attribute name="font-weight">bold</xsl:attribute>
</xsl:attribute-set>

<xsl:template match="chapter/heading">
  <fo:block xsl:use-attribute-sets="title-style">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

</xsl:stylesheet>
```

See also: xsl:attribute, xsl:element, xsl:copy

xsl:call-template

```
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>
```

Invokes a template having a specific name.

This element is defined in Section 6 of the XSLT 1.0 specification.

Use this element to execute a specific template within the stylesheet. It can be thought of as a procedure call and is useful for situations where a generic template can be re-used by several other templates within the stylesheet.

The name attribute is required. Its value must have a local name and an optional prefix. The prefix must match a namespace declaration currently within scope. The XSL processor searches for an xsl:template having a matching name. If no template is found then the XSL processor raises an exception. If more than one matching template is found then the processor chooses the one with the highest import precedence. For information on import precedence, see “Handling template collisions” on page 228.

This element may contain zero or more xsl:with-param elements. The parameters specified by the xsl:with-param elements are visible to the template that is invoked by the XSL processor. Each xsl:with-param element is evaluated only if the template has a matching xsl:param element. If the template contains an xsl:param element that does not have a corresponding xsl:with-param element then the XSL processor uses the default value of the xsl:param element. When the xsl:template is executed, it has the same context as the xsl:call-template element.

The following stylesheet uses a template to pad the names of all contacts in an XML document:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template name="RPadDots">
    <xsl:param name="aString"/>
    <xsl:param name="aLength"/>
    <xsl:variable name="30dots">
      <xsl:text>..............................</xsl:text>
    </xsl:variable>
    <xsl:value-of
     select="substring(concat($aString,$30dots),1,$aLength)"/>
</xsl:template>

<xsl:variable name="padNames">
  <xsl:call-template name="RPadDots">
    <xsl:with-param name="aString"
      select="//contact/@name"/>
    <xsl:with-param name="aLength" select="30"/>
  </xsl:call-template>
</xsl:variable>
```

```
<xsl:template match='/'>
  <xsl:value-of select="$padNames"/>
</xsl:template>

</xsl:stylesheet>
```

See also: xsl:param, xsl:template, xsl:with-param

xsl:choose

```
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
```

Selects one of a number of choices based upon certain criteria.

This element is defined in Section 9.2 of the XSLT 1.0 specification.

Use this element to perform decision-making within a stylesheet. The xsl:choose element is similar to a case statement in Object Pascal or a switch statement in C. The body of the xsl:choose element contains one or more xsl:when elements and an optional xsl:otherwise element. If an xsl:otherwise element is present, it must be the last element. If only one xsl:when element is present and no xsl:otherwise element is present then the xsl:choose behaves as an If statement in other programming languages.

The XSL processor evaluates each xsl:when element from first to last until it encounters an xsl:when element that evaluates to a boolean True. The body of that xsl:when element is then instantiated. If no xsl:when element evaluates to True and an xsl:otherwise element is present, the body of the xsl:otherwise element is instantiated. If no xsl:otherwise element is present then nothing is instantiated.

The following example uses xsl:choose to convert a three letter day code into the name of the day:

```
<xsl:choose>
  <xsl:when test="@dayCode='MON'">Monday</xsl:when>
  <xsl:when test="@dayCode='TUE'">Tuesday</xsl:when>
  <xsl:when test="@dayCode='WED'">Wednesday</xsl:when>
  <xsl:when test="@dayCode='THU'">Thursday</xsl:when>
  <xsl:when test="@dayCode='FRI'">Friday</xsl:when>
  <xsl:when test="@dayCode='SAT'">Saturday</xsl:when>
  <xsl:when test="@dayCode='SUN'">Sunday</xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="@dayCode">
  </xsl:otherwise>
</xsl:choose>
```

See also: xsl:if, xsl:otherwise, xsl:when

xsl:comment

```
<xsl:comment>
  <!-- Content: template -->
</xsl:comment>
```

Creates a comment node in the result tree.

This element is defined in Section 7.4 of the XSLT 1.0 specification.

Use this element to place a comment in the result tree. The template of the xsl:comment element may contain text and elements that result in the creation of text nodes. If the template results in the creation of any other type of node, the XSL processor ignores the nodes. The comment may not contain the string "--" and may not end with the character '-'. If the XSL processor detects either of these situations, it replaces the "--" string with '-' and the '-' character with a space.

This element has no affect on the RTF and Printer filters. However, it causes the HTML and XML filters to embed comments within the output document. The following example embeds the document author and creation date within the XML document:

```
<xsl:comment>
  <xsl:text>Author: </xsl:text>
  <xsl:value-of select="/DocSummary/SourceInfo/@author"/>
  <xsl:text>  Created: </xsl:text>
  <xsl:value-of select="/DocSummary/SourceInfo/@creationDate"/>
</xsl:comment>
```

8

xsl:copy

```
<xsl:copy use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>
```

Performs a shallow copy of the current node to the result tree.

This element is defined in Section 7.5 of the XSLT 1.0 specification.

Use this element to copy the current node to the result tree. This element is useful when the result tree is output in XML and the elements in the input document are to be transferred as is to the output document. Note that xsl:copy performs a shallow copy that excludes the current node's child elements and attributes, if present. To perform a deep copy, use the xsl:copy-of element.

If the current node is the document root, it is not copied to the result tree because the result tree will already have a document root. However, the template body of the xsl:copy element is instantiated in the result tree.

If the current node is an element then xsl:copy places a copy of the element in the result tree. The copy of the element bears the same name as the original. To create an element with a different name, use the xsl:element element. The copy does not include the attributes or children of the original. It does include the namespace nodes of the original. To include attributes, use the use-attribute-sets property to incorporate one or more attribute sets in the copy of the current node. The use-attribute-sets property must be assigned a list of white-space separated names matching attribute sets defined in the stylesheet.

If the current node is a comment, text, or processing instruction node then a new node of the same type and containing the same information is created in the result tree. The use-attribute-sets attribute and template body are ignored.

If the current node is an attribute then an attribute bearing the same name and value is added to the result tree. The copy also includes the namespace nodes associated with the original attribute. The use-attribute-sets attribute and template body are ignored.

The following template copies all attributes of an Invoice node to current node in the result tree:

```
<xsl:template match="invoice">
  <xsl:for-each select="@*">
    <xsl:copy/>
  </xsl:for-each>
</xsl:template>
```

See also: xsl:attribute-set, xsl:copy-of, xsl:element

xsl:copy-of

```
<xsl:copy-of select = expression />
```

Performs a deep copy of the one or more nodes.

This element is defined in Section 11.3 of the XSLT 1.0 specification.

Use this element to copy a document fragment or node set into the result tree. To perform a shallow copy, see the xsl:copy element.

The select attribute specifies the nodes to copy via an XPath expression. If the expression returns a document fragment, the entire fragment is copied to the result tree. If the expression returns a node set then the nodes are copied to the result tree in document order, including their attributes, namespace nodes, and descendants. If the expression returns the document root then the root's descendants are copied to the result tree. If the expression returns any other type of result then the result is converted to a string and copied into the result tree, just as though xsl:value-of were used.

The following example copies an invoice element and its sub-elements (e.g., line items) to the result tree:

```
<xsl:template match="invoice">
  <xsl:copy-of select = "."
</xsl:template>
```

See also: xsl:copy, xsl:value-of

xsl:decimal-format

```
<xsl:decimal-format
  name = qname
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />
```

Defines the characters and symbols used by the format-number function.

This element is defined in Section 12.3 of the XSLT 1.0 specification.

Use this element to control the characters and symbols used when converting numbers to strings using the format-number function. It is ideal for tailoring output of an XML document using the localized numbering style. This element does not affect the behavior of the xsl:number element, the string() function, or the xsl:value-of element.

This element must appear as a top-level element (i.e., a child of xsl:stylesheet). The name attribute is optional. If the name attribute is not present then the xsl:decimal-format element is used as the default decimal format. Only one default decimal format may appear within the stylesheet unless each declaration has the same attribute values.

If the name attribute is present then it must be a valid qualified name and the element is considered a named decimal format. A named decimal format is used only when its name is specified as the third argument of the format-number function. A named decimal format may appear more than once in a stylesheet only if the attribute values among each instance are identical.

If the declaration of a named decimal format does not include an attribute, the XSL processor uses the attribute value from the default decimal format. The meanings of the other attributes are listed in the following table:

Attribute	Type	Description
decimal-separator	Character	Specifies the character used for the decimal sign. The default value is a period character (i.e., '.').
grouping-separator	Character	Specifies the character used as a grouping (e.g., thousands) separator. The default value is a comma character (i.e., ',').
infinity	String	Specifies the string used to represent infinity. The default value is the string "Infinity".
minus-sign	Character	Specifies the character used as the default minus sign. The default value is the hyphen character (i.e., '-').
NaN	String	Specifies the string used to represent NaN (i.e., Not a Number). The default value is the string "NaN".
percent	Character	Specifies the character used as a percent sign. The default value is the percent character (i.e., '%').
per-mille	Character	Specifies the character used as a per mille sign. The default value is the Unicode per-mille character (i.e., #x2030).
zero-digit	Character	Specifies the character used as the digit zero. The default value is the digit zero (i.e., '0').
digit	Character	Specifies the character used for a digit in the format pattern. The default value is the number sign character (i.e., '#').
pattern-separator	Character	Specifies the character used to separate positive and negative sub patterns in a pattern. The default value is the semi-colon character (i.e., ';').

By default, the XSL processor outputs numbers in the format commonly used within the United States. However, this may not be suitable for output being viewed by Europeans. The following example uses an xsl:decimal-format element to define a format using European standards:

```
<xsl:decimal-format
  name="European decimal format"
  decimal-separator=","
  grouping-separator="." />
```

Formatting the number 9,384.55 using “format-number(9384.55, “##.###,##”, ‘European decimal format’)” would output the number “9.3894,55” to the result tree.

xsl:element

```
<xsl:element
  name = qname
  namespace = uri-reference
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>
```

8

Instantiates an element within the result tree.

This element is defined in Section 7.1.2 of the XSLT 1.0 specification.

Use this element to create an element having a specific name within the result tree. Use the template body of the element to create the attributes and children of the new element.

The name of the new element is calculated using the name attribute. The name attribute is required and may take the form of an explicit attribute value or *attribute value template*. An attribute value template is an XPath expression nested within curly braces (i.e., ‘{’ and ‘}’). If the expression does not resolve to a valid qualified name then the XSL processor raises an exception.

The namespace attribute is optional. If not present, the element’s name is expanded using the namespace declarations currently in scope. If the namespace attribute is present, it may take the form of an explicit URI reference or an attribute value template. The resulting string should be a valid URI reference. However, the XSL processor does not check the validity of the string. The XSL processor includes a namespace node within the element’s start tag.

To include predefined attributes within the element start tag, use the use-attribute-sets attribute. The value of use-attribute-sets is a white-space separated list of attribute set names. The attribute sets are included in the element start tag in the order they are listed. To add attributes not contained within an attribute set, use the xsl:attribute element within the body of xsl:element.

The following stylesheet transforms a list of computer elements into a list of elements based upon the computer type:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="computer">
  <xsl:element
    name="{@type}">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet is applied to the following XML document:

```
<?xml version="1.0"?>
<computers>
  <computer type="server">
    <model>Husky 1000</model>
  </computer>
  <computer type="workstation">
    <model>Mongrel</model>
  </computer>
  <computer type="workstation">
    <model>Busboy 2500</model>
  </computer>
</computers>
```

The resulting XML output is as follows:

```
<server>Husky 1000</server>
<workstation>Mongrel</workstation>
<workstation>Busboy 2500</workstation>
```

See also: xsl:attribute, xsl:attribute-set, xsl:copy, xsl:copy-of

```
<xsl:fallback>
  <!-- Content: template -->
</xsl:fallback>
```

Provides a path for alternative processing when an unsupported XSLT or vendor-specific instruction is encountered.

This element is defined in Section 15 of the XSLT 1.0 specification.

If the XSL processor encounters an XSLT or vendor-specific element that it does not recognize, it raises an exception. Use this element to avoid the exception. If the template body of an unsupported instruction contains an xsl:fallback element then the XSL processor executes the xsl:fallback element instead of raising an exception. If the template contains more than one xsl:fallback element then the XSL processor executes them in the order they are specified.

The xsl:fallback template body may be empty in which case the XSL processor continues execution. The xsl:fallback template body may contain other elements and text data which are processed as necessary by the XSL processor.

Instead of using xsl:fallback, the stylesheet could use the element-available function to determine if the XSL processor supports an XSLT or vendor-specific instruction.

The following stylesheet uses the xsl:document element which is unavailable in XSLT 1.0. If the xsl:document element is not supported by the XSL processor then the xsl:fallback element emits a message. Note that the xsl:document element should never be used in a stylesheet where the version number of the xsl:stylesheet is set to “1.0.” However, this was done to illustrate a situation requiring the use of xsl:fallback.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:document href="test.xml" method="xml">
    ...
    <xsl:fallback>
      <xsl:message>
        xsl:document not supported by this XSL processor.
      </xsl:message>
    </xsl:fallback>
  </xsl:document>
</xsl:template>

</xsl:stylesheet>
```

```
<xsl:for-each
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>
```

Instantiates a template for each node in a node set.

This element is defined in Section 8 of the XSLT 1.0 specification.

Use this element to iteratively process a set of nodes. The select attribute is required and must contain an XPath expression that describes a set of nodes. If the expression returns any other type of value then the XSL processor raises an exception.

If the resulting node set does not contain any nodes, the XSL processor ignores the remainder of the xsl:for-each element. If the node set contains one or more nodes, the XSL processor instantiates the template body for each node.

The nodes in the node set are processed in document order, unless one or more xsl:sort elements are used to alter their order. As the template body is instantiated for each node, the selected node becomes the current node and the node set becomes the current node list.

In the following example, the XML document contains a list of cities:

```
<?xml version="1.0"?>
<cities>
  <city name="Colorado Springs" state="Colorado"/>
  <city name="Evansville" state="Indiana"/>
  <city name="St. Louis" state="Missouri"/>
  <city name="Owensboro" state="Kentucky"/>
</cities>
```

The following stylesheet builds an HTML table containing one row per city. The first cell of each row contains the name of the city. The second cell contains the state in which the city is located.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="cities">
  <table border="1">
    <xsl:for-each select="city">
      <tr>
        <td><xsl:value-of select="@name"/></td>
        <td><xsl:value-of select="@state"/></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

8

When the stylesheet is applied to the XML document, the following HTML markup is created:

```
<table border="1">
  <tr>
    <td>Colorado Springs</td><td>Colorado</td>
  </tr>
  <tr>
    <td>Evansville</td><td>Indiana</td>
  </tr>
  <tr>
    <td>St.Louis</td><td>Missouri</td>
  </tr>
  <tr>
    <td>Owensboro</td><td>Kentucky</td>
  </tr>
</table>
```

See also: xsl:sort

```
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>
```

Instantiates a template if a boolean expression evaluates to the value True.

This element is defined in Section 9.1 of the XSLT 1.0 specification.

The xsl:if element is analogous to an IF statement in other programming languages. However, xsl:if does not provide an ELSE clause. Use the xsl:choose element when an ELSE condition is required.

The test attribute of xsl:if is required and must contain an expression that evaluates to the value True or False. Any XPath expression can be converted to True or False, as described in the following table:

Expression Result Type	Conversion to Boolean Type
document fragment	Converts to True if, when converted to a string, the string is not empty.
node set	Converts to True if the node set contains at least one node.
number	Converts to True if the number is non-zero.
string	Converts to True if the string is not empty.

If the test expression evaluates to True then the element's template is instantiated. Otherwise, the XSL processor ignores the body of the xsl:if element.

The following stylesheet builds an HTML table that lists all the line items on a particular order. The template for the lineitem element uses the xsl:if element to set the background color of every other row to yellow.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <table border="1">
  <tr>
    <td>OEM</td>
    <td>Item</td>
    <td>Qty</td>
    <td>Price</td>
  </tr>
  <xsl:apply-templates select="//lineitem"/>
  </table>
</xsl:template>

<xsl:template match="lineitem">
  <tr>
    <xsl:if test="position() mod 2 = 1">
      <xsl:attribute name="bgcolor">yellow</xsl:attribute>
    </xsl:if>
    <xsl:apply-templates select="OEM | descrip | qty | price"/>
  </tr>
</xsl:template>

<xsl:template match="OEM | descrip | qty | price">
  <td>
    <xsl:value-of select="."/>
  </td>
</xsl:template>

</xsl:stylesheet>
```

The following code is when it is applied to the following XML document:

```
<?xml version="1.0"?>
<order>
  <lineitem>
    <OEM>KA445-12</OEM>
    <descrip>Staple dismantler</descrip>
    <qty>4</qty>
    <price>12.00</price>
  </lineitem>
  <lineitem>
    <OEM>3440112</OEM>
    <descrip>Fireproof filing cabinet</descrip>
    <qty>1</qty>
    <price>345.00</price>
  </lineitem>
  <lineitem>
    <OEM>RM44</OEM>
    <descrip>Manager-proof manila folders</descrip>
    <qty>144</qty>
    <price>15.75</price>
  </lineitem>
</order>
```

The resulting HTML markup is as follows:

```
<table border="1">
  <tr>
    <td>OEM</td>
    <td>Item</td>
    <td>Qty</td>
    <td>Price</td>
  </tr>
  <tr bgcolor="yellow">
    <td>KA445-12</td>
    <td>Staple dismantler</td>
    <td>4</td>
    <td>12.00</td>
  </tr>
  <tr>
    <td>3440112</td>
    <td>Fireproof filing cabinet</td>
    <td>1</td>
    <td>345.00</td>
  </tr>
  <tr bgcolor="yellow">
    <td>RM44</td>
```

```
      <td>Manager-proof manila folders</td>
      <td>144</td>
      <td>15.75</td>
    </tr>
  </table>
```

See also: xsl:choose

xsl:import

```
<xsl:import href = uri-reference />
```

Imports a stylesheet into the current stylesheet.

This element is defined in Section 2.6.2 of the XSLT 1.0 specification.

Use this element to import a stylesheet into the current stylesheet. The xsl:import element is just like the xsl:include element except that definitions and template rules in the importing stylesheet take precedence over those in the imported stylesheet.

The xsl:import element must be a top-level element having an xsl:stylesheet element as its immediate parent. A stylesheet may have zero or more xsl:import elements. However, they must all occur before any other elements, including xsl:include elements, within the stylesheet.

The href attribute is required and must contain a valid reference to the stylesheet being imported. The value of href may be absolute or relative. If relative then it is resolved relative to the base URI of the importing stylesheet. If a stylesheet directly or indirectly imports itself, the XSL processor raises an exception. However, any individual stylesheet may be imported in multiple places. The import tree will have a separate entry for each xsl:stylesheet used to import the stylesheet.

Imported stylesheets form an *import tree* and are assigned an *import precedence* based upon their position within that tree. Definitions and templates within a stylesheet having higher import precedence override those definitions and templates within a stylesheet of lower import precedence. Stylesheet A is defined as having lower import precedence than stylesheet B if stylesheet A is visited before stylesheet B during a post-order traversal of the import tree. Figure 8.1 illustrates the import tree created for a number of stylesheets.

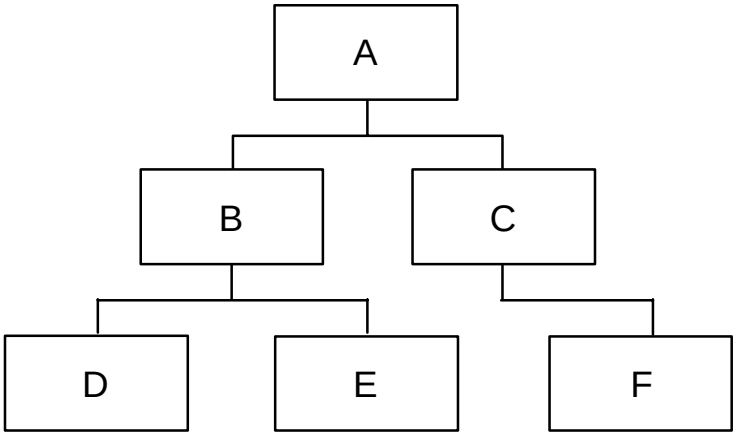


Figure 8.1: Stylesheet import tree

Stylesheet A imports stylesheets B and C. Stylesheets B and C import D, E, and F. Stylesheet A has the highest import precedence because it directly or indirectly imports all of the other stylesheets. The order of import precedence, from highest to lowest, is A, C, F, B, E, D.

Template rules may be overridden by importing a stylesheet containing a template matching the same set of nodes. To invoke the overridden template rules, use the xsl:apply-imports element.

The following stylesheet imports the toc.xsl stylesheet and includes the contents.xsl stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:import href="toc.xsl"/>
<xsl:include href="contents.xsl"/>
...
</xsl:stylesheet>
```

See also: xsl:apply-imports, xsl:include

xsl:include

```
<xsl:include href = uri-reference />
```

Includes a stylesheet in the current stylesheet.

This element is defined in Section 2.6.1 of the XSLT 1.0 specification.

Use this element to include a stylesheet within another stylesheet. The xsl:include element is just like the xsl:import element except that definitions and template rules within the included stylesheet are given the same import precedence as the including stylesheet.

The xsl:include element is a top-level element that must have an xsl:stylesheet element as its immediate parent. A stylesheet may have zero or more xsl:include elements. However, they must occur after any xsl:import elements and before the remaining elements within the stylesheet.

The href attribute is required and must contain a valid reference to the included stylesheet. The value of href may be absolute or relative. If relative then it is resolved relative to the base URI of the including stylesheet. If a stylesheet directly or indirectly includes itself, the XSL processor raises an exception. However, any individual stylesheet may be included in multiple places. Note that while this is valid, it may lead to unexpected errors due to duplicate definitions.

The following stylesheet imports the toc.xsl stylesheet and includes the contents.xsl stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:import href="toc.xsl"/>
<xsl:include href="contents.xsl"/>
...
</xsl:stylesheet>
```

See also: xsl:import

xsl:key

```
<xsl:key
  name = qname
  match = pattern
  use = expression />
```

Declares a named key used to index a subset of nodes within a document.

This element is defined in Section 12.2 of the XSLT 1.0 specification.

Use this element when you need to quickly access nodes within an XML document based upon a key value. XML provides a way to uniquely identify elements using an ID attribute. However, an ID attribute has several limitations:

- If the ID attribute is declared in an external DTD then the XML processor may ignore the external DTD.
- The XML document may have only one set of unique IDs.
- An ID is constrained to be an XML name. For example, it cannot contain spaces.
- An element can have at most one ID.
- Non-unique IDs may not be used.

To overcome these limitations, XSLT provides the xsl:key element. This element generates an index of keys for a set of nodes matching a specific pattern. A key identifies a particular node having a specific key with a specific value. The xsl:key element is a top-level element, meaning that it must have an xsl:stylesheet element as its immediate parent.

A document may contain zero or more instances of the xsl:key element. This means that you may index the nodes in a document in multiple ways. The name attribute of the xsl:key element is required and specifies the name of the key. If the name contains a namespace prefix then it must match with a namespace currently in effect for the xsl:key element.

The match attribute is a pattern identifying the nodes for which a key may be generated. The use attribute is a XPath expression used to generate the key value. The following xsl:key element defines a key on a product's Original Equipment Manufacturer (OEM) number:

```
<xsl:key name="OEM" match="product" use="@OEM"/>
```

To access a specific product by its OEM number, the stylesheet could use the XSLT key() function as follows:

```
<xsl:template match="StockingLocation">
  <xsl:for-each select="key('OEM',@ProductOEM)">
    <xsl:value-of select="@name"/>
  </xsl:for-each>
</xsl:template>
```

When the stylesheet encounters a StockingLocation element, it locates the product element using the key value contained in the ProductOEM attribute of the StockingLocation element. The XSL processor searches for the product having that OEM number. If found, it outputs the value of the product element's name attribute to the result tree.

If two or more xsl:key elements have the same key name, the node sets of each xsl:key element are merged together. A node may have multiple key values. If the use attribute returns anything other than a node set then the key is generated by converting the result to a string. If the use attribute returns a node set, a key is generated for each node in the set by converting the node to a string value.

The xsl:key element supports non-unique keys. If a key contains non-unique keys then the key() function returns all nodes with matching key values.

xsl:message

```
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>
```

Causes the XSL processor to raise its OnXSLMessage event.

This element is defined in Section 13 of the XSLT 1.0 specification.

Use this element to output a message to the application using the XMLPartner XSL processor. This element is useful for reporting errors detected by logic within the stylesheet itself. If the TXpXSLProcessor component has an event handler for its OnXSLMessage event then that event handler is called. The message passed to the event handler is generated using the template body of the xsl:message element. The contents of the template are converted to a string and passed in the Msg parameter of the event handler.

The terminate attribute defaults to “no” and indicates whether the XSL processor should stop processing the stylesheet. When the OnXSLMessage event handler is called, the value of the terminate attribute is passed in the event handler’s Terminate parameter. The event handler may override the value of the Terminate parameter if it so desires. If the terminate attribute is set to “no” then processing continues normally. If the terminate attribute is set to “yes” then the XSL processor stops processing the stylesheet.

The following stylesheet uses the xsl:document element which is unavailable in XSLT 1.0. If the xsl:document element is not supported by the XSL processor then the xsl:fallback element emits a message and terminates the processing of the stylesheet.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:document href="test.xml" method="xml">
    ...
    <xsl:fallback>
      <xsl:message terminate="yes">
        xsl:document not supported by this XSL processor.
      </xsl:message>
    </xsl:fallback>
  </xsl:document>
</xsl:template>

</xsl:stylesheet>
```


xsl:namespace-alias

```
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

Maps a stylesheet namespace to a different namespace in the result tree.

This element is defined in Section 7.1.1 of the XSLT 1.0 specification.

Use this element to declare an alias for a namespace. Each occurrence of the alias in the stylesheet is translated into another namespace in the result tree and final output. One of the uses of this element is to create a stylesheet that generates another stylesheet. For example, if you want to generate a stylesheet that contains an xsl:template element then you cannot put xsl:template into the source stylesheet because it would cause the XSL processor to generate an error.

The stylesheet-prefix and result-prefix attributes are both required. The stylesheet-prefix attribute identifies the prefix of the namespace that is being aliased. The result-prefix identifies the prefix of the namespace to which the stylesheet-prefix is being translated in the result tree and final output. To specify the default namespace for either attribute, use the string “#default”.

The following stylesheet uses the xsl:namespace-alias element to generate a new stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:test="testnamespace"
  version="1.0">

<xsl:namespace-alias stylesheet-prefix="temp" result-
prefix="xsl"/>

<xsl:template match="/">
  <temp:stylesheet version="1.0">
    <temp:template match="/">
      <temp:value-of select="."/>
    </temp:template>
  </temp:stylesheet>
</xsl:template>

</xsl:stylesheet>
```


The stylesheet elements to be output to the new stylesheet are prefixed with “temp” instead of “xsl”. The xsl:namespace-alias element maps the “temp” namespace to the “xsl” namespace. When applied to any XML document, the resulting output document is as follows:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Note that xsl:namespace-alias is a top-level element and must have an xsl:stylesheet element as its immediate parent. A stylesheet may contain more than one xsl:namespace-alias element. If a document contains more than one xsl:namespace-alias element having the same value for their stylesheet-prefix attribute then the XSL processor uses the xsl:namespace-alias element with the highest import precedence. If it finds two or more xsl:namespace-alias elements having the highest import precedence then the behavior of the XSL processor depends upon the RaiseConflictError property of the TXpXSLProcessor component. If RaiseConflictError is set to True then the XSL processor raises an exception. Otherwise, the XSL processor chooses the last declared xsl:namespace-alias among those with the highest import precedence.

See also: TXpXSLProcessor.RaiseConflictError

xsl:number

```
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />
```

Generates a sequence number or formats a specific number.

This element is defined in Section 7.7 of the XSLT 1.0 specification.

This element may be used to either format a value for output or to generate sequence numbers for one or more nodes. The first part of this section describes the formatting of a specific value. The remainder of the section describes the creation of sequence numbers.

Formatting a number

When formatting a number, the value attribute must be assigned an expression. The expression is evaluated and the resulting object is converted to a number. The number is rounded to an integer and converted to a string. The string is then written to the result tree. For example, the following template rounds a price attribute to an integer value and uses the period character as a grouping separator:

```
<xsl:number value="@price" grouping-separator="."/>
```

To control the format of the resulting number, use the following attributes:

Attribute	Type	Purpose
format	Attribute value template returning a format string.	Specifies the manner in which the number is written to the result tree. This attribute is optional and defaults to the string “1”. The format attribute is described in further detail following this table.
grouping-separator	Attribute value template that returns a single character.	Specifies the character used to separate groups (e.g., thousands). This attribute is optional and defaults to the separator for the computer’s locale.
grouping-size	Attribute value template that returns a number.	Specifies the number of digits within each grouping (e.g., hundreds, thousands). The character specified by the grouping-separator attribute is inserted between each group. This attribute is optional and defaults to the value 3.
lang		This attribute is part of the XSLT 1.0 specification but is ignored by the XSL processor.
letter-value		This attribute is part of the XSLT 1.0 specification but is ignored by the XSL processor.

The count, from, and level attributes do not affect the formatting of specific values. Use the format attribute to control the formatting of the number in the result tree. The format string is comprised of a series of alphanumeric tokens with tokens separated by one or more non-alphanumeric characters. The alphanumeric value of a token affects the formatting of the number, as shown in the following table:

Token	Result
1	Output as number (e.g., 1, 5, 120).
A	Output as uppercase alphabetic (e.g., A, B, Z).
a	Output as lowercase alphabetic (e.g., a, b, z).
i	Output as lowercase Roman numeral (e.g., i, ii, iv).
I	Output as uppercase Roman numeral (e.g., I, II, IV).

Non-alphanumeric characters before or after a token are output as is to the result tree. The following table lists several examples of using this element to format a number:

Example	Result
`<xsl:number value="10" format="I"/>`	‘X’
`<xsl:number value="2" format="a"/>`	‘b’
`<xsl:number value="1234567" grouping-separator="|" grouping-size=2"/>`	“1|23|45|76”
`<xsl:number value="6" format="A."/>`	“G.”

8

Generating sequence numbers

If the value attribute is omitted from the xsl:number element then the XSL processor generates a sequence number for the current node. Note that sequence numbers may be generated for any type of node, not just element nodes. The count, from, and level attributes influence the assignment of sequence numbers. Each of these attributes is optional and they are summarized in the following table:

Attribute	Meaning
count	If specified, this is a pattern that controls what nodes are included in the generation of the sequence number. If namespaces are being used, the name comparison is performed using the namespace URI, not the prefix. If this attribute is not specified, the XSL processor matches nodes having the same type and the same expanded name, if applicable.
from	Specifies where counting starts. By default, the first sequence number is 1. Use this attribute to set a higher starting point, such as 10.
level	Controls the way nodes are numbered based upon their position within the XML document. The default value of this attribute is “single”.

There are at least three scenarios in which nodes need to be numbered. For example, a list of items may need to be assigned sequential numbers. The footnotes spread throughout a chapter may need to be assigned a sequential number. Or the topic elements within a document may need to be structured as an outline (e.g., 1.1, 1.2, 1.3). Use the level attribute to control the numbering style. The legal values for the level attribute are described in the following table:

Level Value	Purpose
any	Use this value to generate a sequence number for nodes that can appear at any level of a document. For example, this is useful to generate identifiers for tables, figures, or footnotes within a chapter.
multiple	Use this value to generate a sequence number for nodes based upon their position in a document. For example, this is useful to generate identifiers for chapters and sections within chapters.
single	Use this value to generate a sequence number for a contiguous set of nodes. For example, this is useful to generate numbers for a list of items.

The simplest way to use this element is as shown in the following stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="state | city">
  <xsl:value-of select="@name"/>
  <xsl:text>...</xsl:text>
  <xsl:number/>
  <xsl:text>
  </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

The stylesheet is applied to the following XML document:

```
<?xml version="1.0"?>
<cities>
  <state name="Colorado">
    <city name="Colorado Springs"/>
    <city name="Denver"/>
    <city name="Divide"/>
    <city name="Pueblo"/>
  </state>
</cities>
```

The resulting output is as follows:

```
Colorado...1
Colorado Springs...1
Denver...2
Divide...3
Pueblo...4
```

The state element is assigned the number 1 because it is the first state element in the document. The city elements are also assigned their respective number based upon their position relative to their sibling city elements.

For a more complex example, consider the situation where a manual is written to an XML document. The manual is divided into chapters. Each chapter contains one or more primary elements (i.e., primary sections of the manual) and each primary element contains one or more secondary elements (i.e., secondary sections of the manual). The following template produces an outline of the document based upon its chapter, primary, and secondary elements:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/ | manual">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="chapter | primary | secondary">
  <xsl:number count="chapter | primary | secondary"
              format="1.A.i  "
              level="multiple"/>
  <xsl:value-of select="@title"/>
  <xsl:text>
  </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

8

xsl:otherwise

```
<xsl:otherwise>
  <!-- Content: template -->
</xsl:otherwise>
```

Specifies the template to be instantiated within an xsl:choose element when none of the xsl:when conditions evaluate to the value True.

This element is defined in Section 9.2 of the XSLT 1.0 specification.

The xsl:choose element evaluates each xsl:when element until it finds one that evaluates to True. If none of the xsl:when elements evaluate to True and an xsl:otherwise element is present then the XSL processor instantiates the body of the xsl:otherwise element.

This element must have an xsl:choose element as its immediate parent. It may not appear in any other content. An xsl:choose element may have only one child xsl:otherwise element.

In the following example, the template decides what tax information to display in an invoice based upon whether or not the customer has a taxable status:

```
<xsl:template match="customer">
  <xsl:choose>
    <xsl:when test="@taxStatus='N/A'">
      <xsl:text>No tax charged.</xsl:test>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>Tax rate: </xsl:text>
      <xsl:value-of select="@taxRate"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

See also: xsl:choose, xsl:when

xsl:output

```
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />
```

Writes a result tree to an output document in a specific format.

This element is defined in Section 16 of the XSLT 1.0 specification. Please see the README.HLP or README file accompanying this product for TurboPower-specific extensions to this element.

XMLPartner Professional provides two mechanisms for saving a result tree to disk. The first is through the use of a filter component. The second is through the use of an xsl:output element. The xsl:output element tells the XSL processor to write the result tree to a local file. If your stylesheet contains an xsl:output element and a filter is not connected to the TXpXSLProcessor component then the XSL processor writes to the file as directed by the xsl:output element. Connecting a filter to the TXpXSLProcessor component overrides the xsl:output element.

Use the method attribute to control the format of the output document. The value “xml” causes the result tree to be output as an XML document. The result tree can be output as HTML using the value “html” and as text using the value “text.” XMLPartner Professional recognizes additional values for this attribute in order to support output in other formats. Please see the README.HLP or README file accompanying this product for more information.

If the method attribute is not present, the XSL processor determines the output format based upon several conditions. The result tree is output as “html” if the following conditions are True:

- The document root of the result tree has an element child.
- The first element child of the document root has an expanded name with local part equal to “html” (case-insensitive) and a null namespace URI.
- All nodes preceding the first element child contain only white-space characters.

Otherwise, the result tree is output as “xml”. When the stylesheet is processed, it is not immediately written to disk. In order to write the result tree to disk, the SaveToFile method of the TXpXSLProcessor component must be called.

The other attributes supported by the xsl:output element are optional and their use depends upon the value of the method attribute. The attributes and their values are described in the following sections.

The xsl:output element is a top-level element and must have an xsl:stylesheet element as its immediate parent. There may be more than one xsl:output element in a stylesheet and in any stylesheets included or imported by the stylesheet. All xsl:output elements in a stylesheet are merged into a single xsl:output element. The value of the cdata-section-elements is the union of all cdata-section-elements attributes from each xsl:output element. The value of the method attribute and all other attributes is obtained from the xsl:output element with the highest import precedence.

XML output

The “xml” output method outputs the result tree as well-formed XML. Depending upon the values assigned to the xsl:output attributes, the XML may take the form of a document or a document fragment. The following table lists the behavior of the xsl:output attributes when the method attribute is assigned the value “xml”:

Attribute	Data Type	Purpose
cdata-section-elements	White-space separated list of qualified names.	Identifies the elements that should be output as CDATA sections.
doctype-public	String	This attribute is used only if a value is supplied for the doctype-system attribute. If a value is supplied for this attribute then a PUBLIC identifier is written to the document type declaration. See the doctype-system attribute for more details.
doctype-system	String	This attribute causes a document type declaration, which refers to an external DTD, to be written to the output document. If a value is specified for this attribute then a document type declaration is written to the output document immediately before the first element. The name following “<!DOCTYPE” is the name of the first element. If a value is specified for the doctype-public attribute then the string “PUBLIC” followed by the public identifier is written to the output document. The string “SYSTEM” followed by the system identifier is then written to the output document.

Attribute	Data Type	Purpose
encoding	String	Specifies the character set in which the output document is written. The XSL processor recognizes the values “UTF-8”, “UTF-16” and “ISO-8859-1”. If a value is not supplied for this attribute then the XSL processor defaults to the character encoding of the original XML document.
indent	“yes” | “no”	If the value “yes” is assigned to this attribute then the XSL processor will indent the output document to improve readability. The default value is “no”.
media-type	String	This attribute is ignored by the XSL processor.
omit-xml-declaration	“yes” | “no”	If the value “yes” is assigned to this attribute then the XSL processor will not write an XML declaration to the output document. The default value is “no”. When writing the XML declaration, the XSL processor includes both the version and encoding attributes. If the standalone attribute is assigned a value then a standalone attribute is also written with the same value.
standalone	“yes” | “no”	If a value is assigned to this attribute then the XML declaration written to the output document, if any, will include a standalone attribute with the same value.

Attribute	Data Type	Purpose
version	NMToken	Specifies the version of XML used in the output document. At the time of this writing, the XSL processor supports XML 1.0. If this attribute is assigned a value other than “1.0” then the XSL processor outputs “1.0” in the XML declaration.

The following example outputs the result tree as an XML document containing the XML declaration as well as a standalone attribute:

```
<xsl:output method="xml"
            omit-xml-declaration="no"
            standalone="yes"/>
```

HTML output

The html output method saves the result tree as an HTML document. When saving to HTML, the XSL processor has the following special behaviors:

- The following elements are written as empty elements without an end tag:

area	frame	link
base	hr	meta
basefont	img	param
br	input	
col	isindex	

- Characters such as less than (<) and ampersand (&) are not escaped when being written in the body of the script and style elements.
- Processing instructions are terminated with the character ‘>’ instead of “?>”.
- Attributes that are written in HTML as a keyword only are written as keywords. For example, “<OPTION selected>” is written instead of “<OPTION selected=‘selected’>”

The following table lists the behavior of the xsl:output attributes for this output method:

Attribute	Data Type	Purpose
cdata-section-elements	White-space separated list of qualified names.	This attribute is ignored by the XSL processor for HTML output.
doctype-public	String	If a value is specified for this attribute then the XSL processor outputs a document type declaration before the first element in the document. The XSL processor then outputs the string "PUBLIC" followed by the specified public identifier.
doctype-system	String	If a value is specified for this attribute then the XSL processor outputs a document type declaration before the first element in the document. The XSL processor then outputs the string "SYSTEM" followed by the specified system identifier. If a value is also supplied for the doctype-public attribute then the string "SYSTEM" and the system identifier are output after the public identifier.
encoding	String	Specifies the character set in which the output document is written. If the document contains a HEAD element then the XSL processor adds a META element immediately after the start tag of the HEAD element. The META element identifies the character encoding being used.
indent	"yes" | "no"	If the value "yes" is assigned to this attribute then the XSL processor will indent the output document to improve readability. The default value is "yes".

Attribute	Data Type	Purpose
media-type	String	This attribute is ignored by the XSL processor.
omit-xml-declaration	“yes” | “no”	This attribute is ignored by the XSL processor for HTML output.
standalone	“yes” | “no”	This attribute is ignored by the XSL processor for HTML output.
version	NMToken	Specifies the version of HTML used in the output document. At the time of this writing, the XSL processor supports HTML 4.0. If this attribute is assigned a value other than “4.0” then the XSL processor outputs “4.0” in the XML declaration.

Text output

The text output method writes the string value of every text node in the result tree to the output document. None of the text node characters are escaped. The following table lists the behavior of the xsl:output attributes for this output method:

Attribute	Data Type	Purpose
cdata-section-elements	White-space separated list of qualified names.	
doctype-public	String	This attribute is ignored by the XSL processor for text output.
doctype-system	String	This attribute is ignored by the XSL processor for text output.
encoding	String	Specifies the character set in which the output document is written. If this attribute is not specified then the encoding defaults to the character encoding of the original XML document.

Attribute	Data Type	Purpose
indent	“yes” | “no”	This attribute is ignored by the XSL processor for text output.
media-type	String	This attribute is ignored by the XSL processor.
omit-xml-declaration	“yes” | “no”	This attribute is ignored by the XSL processor for text output.
standalone	“yes” | “no”	This attribute is ignored by the XSL processor for text output.
version	NMToken	This attribute is ignored by the XSL processor for text output.

xsl:param


```
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>
```

Declares a global parameter or the local parameter of a template.

This element is defined in Section 11 of the XSLT 1.0 specification.

Use this element to declare a global parameter passed into the stylesheet or a local parameter passed to a template. The name attribute is required and identifies the name of the parameter. The select attribute is optional. If specified, it must contain an expression that evaluates to the parameter’s default value. The default value may also be specified via the xsl:param element’s template body. If a select attribute is specified then the xsl:param element must be empty.

Note: The default value, regardless of how it is specified, is used only if the parameter is not passed into the stylesheet or template.

When the xsl:param element is a top-level element within the stylesheet then it is considered a global parameter. The value of the parameter may be passed into the stylesheet through the TXpXSLProcessor component. At the time of this writing, the parameter interface of the TXpXSLProcessor component was not defined. Please review the README or README.HLP file supplied with this installation for further details.

To declare a local parameter for a template, place the xsl:param element as a child of an xsl:template method. An xsl:template may have zero or more xsl:param elements. The xsl:param elements must occur before any other elements within the template.

The following example defines a template that formats a dollar amount for output to the result tree. The template has the name “FormatPrice” and accepts the price in a parameter named “Price.”

```
<xsl:template name="FormatPrice">
  <xsl:param name="Price" select="0"/>
  <xsl:value-of select="format-number($Price,'###.00')"/>
</xsl:template>
```

See also: xsl:template, xsl:variable

xsl:preserve-space

```
<xsl:preserve-space elements = tokens />
```

Preserves whitespace-only text nodes of specific nodes in the source document.

This element is defined in Section 3.4 of the XSLT 1.0 specification.

By default, the XSL processor preserves whitespace-only text nodes in the XML source document. This may be overridden using the xsl:strip-space element. The xsl:preserve-space element is only used to adjust the behavior set by the xsl:strip-space element. While the xsl:strip-space element specifies the names of the elements in which whitespace-only nodes are to be removed, the xsl:preserve-space element specifies the names of elements in which whitespace-only nodes may not be removed.

The elements attribute is required. Its value must be a string containing one or more tokens. Each token must take one of the following forms:

Value	Meaning
*	Preserve whitespace-only nodes in all elements.
prefix:*	Preserve whitespace-only nodes in all elements within a certain namespace.
qname	Preserve whitespace-only nodes in elements having a specific name.

In the following example, whitespace-only nodes are stripped in all elements except those having the name CustomerName or in the namespace “desc”:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="CustomerName desc:*"/>
...
</xsl:stylesheet>
```

Conflicts between xsl:strip-space and xsl:preserve-space elements are resolved by choosing the element with the highest import precedence and priority.

See also: xsl:strip-space

xsl:processing-instruction

```
<xsl:processing-instruction
  name = { ncname }>
  <!-- Content: template -->
</xsl:processing-instruction>
```

Creates a process instruction node in the result tree.

This element is defined in Section 7.3 of the XSLT 1.0 specification.

Use this element to place a processing instruction node in the result tree. This element is useful only for result trees being output as XML or HTML. The name attribute is required and is used as the name of the processing instruction element. It is evaluated as an attribute value template. The name may not contain a colon and it may not be assigned the value “xml” using any combination of upper or lower case letters (e.g., “XML”, “Xml”). If any of these conditions are not met then the XSL processor raises an exception.

The body of the xsl:processing-instruction generates the remainder of the processing instruction. The body of the xsl:processing-instruction may not contain the text string “?>”. If this character sequence is encountered, the XSL processor raises an exception. The XSL processor automatically outputs the end tag of the processing instruction and requires no intervention from the stylesheet.

See also: xsl:output

```
<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" | qname-but-not-cname }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />
```

Specifies the order in which nodes are to be sorted.

This element is defined in Section 10 of the XSLT 1.0 specification.

Use this element to sort the nodes selected by an xsl:apply-templates or xsl:for-each element. When used with an xsl:apply-templates element, the xsl:sort elements may appear before, after, or interspersed with xsl:with-param elements. When used with an xsl:for-each element, the xsl:sort elements must appear before the template body of the xsl:for-each element.

The nodes selected by an xsl:apply-templates or xsl:for-each element are sorted, as specified by the xsl:sort element(s), before the template bodies are instantiated. Each attribute of xsl:sort is optional. The following table lists the purpose of each attribute:

Attribute	Type	Meaning
case-order	Attribute value template	This attribute is used only when the data-type attribute is assigned the value "text". If the value of the attribute resolves to “upper-first”, uppercase characters sort before lowercase characters (e.g., A a B b). If the value resolves to “lower-first”, lowercase characters sort before uppercase characters (e.g., a A b B).
data-type	Attribute value template	If the value of this attribute resolves to “text” then the sort keys are sorted alphabetically. If the value resolves to “number” then the sort keys are sorted numerically. The XSL processor raises an exception if any other value is specified for this attribute. The default value of this attribute is “text.”

Attribute	Type	Meaning
lang	Attribute value template	The XSL processor ignores this attribute. Sorting is performed using the locale of the current user.
order	Attribute value template	If the value of this attribute resolves to “ascending” then the sort keys are sorted in ascending order. If the value resolves to “descending”, the sort keys are sorted in descending order. The default value of this attribute is “ascending.”
select	Expression	If specified, this attribute generates the sort key for each node. The value of the attribute is an expression that is evaluated for each node where the node being evaluated is the current node and the list of nodes is the current node list. The resulting value is converted to a string. The default value of this attribute is ‘.’ which causes the string value of the current node to be used as the sort key.

Both the xsl:apply-templates and xsl:for-each elements may contain one or more xsl:sort elements. The first xsl:sort child is considered to be the primary sort key, the second xsl:sort child is the second sort key, and so on.

In the following example, the XML document contains a list of cities:

```
<?xml version="1.0"?>
<cities>
  <city name="St. Louis" state="Missouri"/>
  <city name="Colorado Springs" state="Colorado"/>
  <city name="Evansville" state="Indiana"/>
  <city name="Kansas City" state="Missouri"/>
  <city name="Owensboro" state="Kentucky"/>
  <city name="Indianapolis" state="Indiana"/>
</cities>
```

The following stylesheet builds an HTML table containing one row per city. The first cell of each row contains the name of the city. The second cell contains the state in which the city is located. The cities template uses two xsl:sort elements. The first sorts the cities by state. The second sorts the cities by their name.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="cities">
  <table border="1">
    <xsl:for-each select="city">
      <xsl:sort select="@state"/>
      <xsl:sort select="@name"/>
      <tr>
        <td><xsl:value-of select="@name"/></td>
        <td><xsl:value-of select="@state"/></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

When the stylesheet is applied to the XML document, the following HTML markup is created:

```
<table border="1">
  <tr>
    <td>Colorado Springs</td><td>Colorado</td>
  </tr>
  <tr>
    <td>Evansville</td><td>Indiana</td>
  </tr>
  <tr>
    <td>Indianapolis</td><td>Indiana</td>
  </tr>
  <tr>
    <td>Kansas City</td><td>Missouri</td>
  </tr>
  <tr>
    <td>St.Louis</td><td>Missouri</td>
  </tr>
  <tr>
    <td>Owensboro</td><td>Kentucky</td>
  </tr>
</table>
```


See also: xsl:apply-templates, xsl:for-each

xsl:strip-space

```
<xsl:strip-space elements = tokens />
```

Strips whitespace-only text nodes from specific nodes in the source document.

This element is defined in Section 3.4 of the XSLT 1.0 specification.

By default, the XSL processor preserves whitespace-only text nodes in the XML source document. To override this behavior, use the xsl:strip-space element to specify the names of elements in which whitespace-only nodes are to be removed. Use the xsl:preserve-space element to adjust the elements affected by xsl:strip-space.

The elements attribute is required. Its value must be a string containing one or more tokens. Each token must take one of the following forms:

Value	Meaning
*	Strip whitespace-only nodes in all elements.
prefix:*	Strip whitespace-only nodes in all elements within a certain namespace.
qname	Strip whitespace-only nodes in elements having a specific name.

In the following example, whitespace-only nodes are stripped in all elements except those having the name CustomerName or in the namespace desc:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="CustomerName desc:*"/>
...
</xsl:stylesheet>
```

Conflicts between xsl:strip-space and xsl:preserve-space elements are resolved by choosing the element with the highest import precedence and priority.

See also: xsl:preserve-space

xsl:stylesheet

```
<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:stylesheet>
```

Represents and declares a stylesheet.

This element is defined in Section 2.2 of the XSLT 1.0 specification.

This element is the outermost element of a stylesheet. The xsl:transform element serves as a synonym for this element, having the same attributes and usage. However, it is more common for a stylesheet to be declared using an xsl:stylesheet element. For a step-by-step guide to creating stylesheets, see “Creating a Stylesheet” on page 217.

A stylesheet may be contained in its own document, in which case the xsl:stylesheet element serves as the root element of the document. Alternatively, the stylesheet may be embedded within an XML document in which case the xsl:stylesheet element is a child of the document's root element.

The version attribute is required and must identify the version of XSLT required by the stylesheet. The XMLPartner XSL processor accepts the value "1.0" for this attribute. Any other value causes an exception to be raised.

The xmlns:xsl attribute is a mandatory namespace declaration that identifies the document as an XSLT stylesheet. The value must be specified exactly as shown in the element declaration. Earlier versions of stylesheets use a different namespace declaration, as shown in the following declaration:

```
xmlns:xsl="http://www.w3.org/TR/WD-xsl"
```

Stylesheets containing this declaration are from an older version of Microsoft XSL that conformed to a draft version of the XSL specification.

The remaining attributes of the xsl:stylesheet element are optional. The id attribute assigns a unique identifier to the stylesheet. This attribute is used when the stylesheet is embedded within an XML document and it is to be invoked via a xsl-stylesheet processing instruction. The XMLPartner XSL processor can invoke an embedded stylesheet without the existence of a xsl-stylesheet processing instruction.

The extension-element-prefixes attribute is a whitespace-separated list of namespace prefixes. At the time of this writing, the XMLPartner XSL processor ignores the value of this attribute.

The exclude-result-prefixes attribute is a whitespace-separated list of namespace prefixes. When the XSL processor copies an element to the result tree, it includes a namespace node for each namespace that is in scope for the element. This could lead to a lot of redundant, useless namespace information. If the result tree is output to an XML document, the namespace nodes don't affect its validity but they may make it difficult to read the document. Use this attribute to prevent specific namespace nodes from being output to the result tree.

An xsl:stylesheet element may contain the following types of elements:

xsl:attribute-set	xsl:key	xsl:preserve-space
xsl:decimal-format	xsl:namespace-alias	xsl:strip-space
xsl:import	xsl:output	xsl:template
xsl:include	xsl:param	xsl:variable

When one of the above elements occurs as a child of the xsl:stylesheet element then that child is referred to as a *top-level* element. The order of the children within an xsl:stylesheet element is insignificant with the exception of xsl:import elements. All xsl:import elements must occur before any other top-level elements. The xsl:stylesheet element may contain other elements that are outside the XSLT namespace. However, the XSL processor ignores those elements.

See also: xsl:import, xsl:transform

xsl:template

```
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- Content: (xsl:param*, template) -->
</xsl:template>
```

Defines a template rule.

This element is defined in Section 5.3 of the XSLT 1.0 specification.

The transformation of an XML document involves filling a result tree with nodes and text. Use the xsl:template element to carry out the creation of the result tree. The xsl:template element is a top-level element, requiring an xsl:stylesheet element as its immediate parent.

The attributes of the xsl:template element are optional. If the match attribute is present, it specifies a pattern that matches zero or more nodes within the document. The XSL processor executes a template when the xsl:apply-templates element is invoked and one or more nodes matching the pattern are found. The match pattern may not reference a variable. If the match attribute is not present, a name attribute must be specified.

If the name attribute is present, it specifies a name for the template. The XSL processor executes the named template when an xsl:call-template element invokes the template by name. This is similar to a subroutine or procedure call in other programming languages. If the name attribute is not present, a match attribute must be specified.

If both a match and name attribute are present, the template may be executed via the xsl:apply-templates or xsl:call-template elements. For examples of each situation, see Chapter 6.

When the XSL processor invokes a template, the processor executes the template's body. The template body may contain other XSLT elements that invoke other templates or perform other tasks causing nodes to be output to the result tree. The template body may also contain literal result elements that are outside the XSLT namespace. Literal result elements are copied to the result tree as is. The template body may also contain text that is transferred to the result tree as text nodes.

The selection of a template may be influenced using the priority and mode attributes. When more than one template matches a given node, the XSL processor uses a conflict resolution algorithm to determine which xsl:template element is invoked. To influence the conflict resolution algorithm, use the priority attribute to improve or lessen the template's chances of being chosen. The priority attribute must be assigned a number that is positive or negative. If the priority is not specified, it defaults to a number based upon the selectivity of the pattern. This logic is defined in Section 5.5 of the XSLT 1.0 specification.

The mode attribute, if present, must be assigned a qualified name. If a template is assigned a mode, it is considered for execution by the XSL processor only when executing an xsl:apply-templates element with a matching node. This means the stylesheet can switch between templates with identical match attributes but differing modes. A template may have a mode attribute only if it has a match attribute.

In the following example, the stylesheet generates an HTML document containing a table of contents followed by the text for each topic:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <xsl:apply-templates select="//topic" mode="toc"/>
  <xsl:apply-templates select="//topic"/>
</xsl:template>

<xsl:template match="topic" mode="toc">
  <A href="#{.}">
  <xsl:value-of select="."/>
  </A>
  <br/>
</xsl:template>

<xsl:template match="topic">
  <H3>
```

```
      <A name="{.}">
      <xsl:value-of select="."/>
      </A>
    </H3>
  </xsl:template>
  </xsl:stylesheet>
```

The template for the document root first applies template rules for all topics elements with mode “toc.” It then applies template rules for all topics without a mode. The template rule for topics with mode “toc” creates a hyperlink to a bookmark created in the template rule for topics without a mode.

The following stylesheet applied to the following XML document:

```
<topics>
  <topic>Customers</topic>
  <topic>Vendors</topic>
</topics>
```

The stylesheet generates the following output:

```
<A href="#Customers">Customers</A><br/>
<A href="#Vendors">Vendors</A><br/>
<H3><A name="Customers">Customers</A></H3>
<H3><A name="Vendors">Vendors</A></H3>
```

Importing and overriding template rules

The xsl:import element allows one stylesheet to import another stylesheet. A template in an importing stylesheet has a higher precedence than a template within an imported stylesheet. More details about import precedence can be found in the section “Importing stylesheets” on page 227 or the xsl:import element reference, on page 316. The import mechanism allows one template to override another, if they have the same match attribute and mode. When selecting a template for a given node, the XSL processor chooses the template with higher import precedence.

The template with lower import precedence may be invoked from the template with higher import precedence using the xsl:apply-imports element. This is similar to overriding a class method in other programming languages.

Handling template conflicts

It is possible for more than one template to match a given node. When an xsl:apply-templates element is executed, the XSL processor considers only those templates that have a match attribute, the same mode as the xsl:apply-templates element, and a pattern matching the node. If more than one template meets each condition, the XSL processor uses the following logic to narrow down the possibilities to one template:

- Select the templates with the highest import precedence.
- If more than one template has the highest import precedence, select the template with the highest priority.
- If more than one template remains, handle the situation based upon the value specified for the RaiseConflictError property of the TXpXSLProcessor component.

Built-in template rules

The XSLT 1.0 specification defines several built-in template rules. Built-in template rules are used when the XSL processor does not find an explicit template matching the selected node. The XSL processor treats built-in template rules as though they were imported explicitly before the stylesheet and so have lower import precedence than all other template rules. The following built-in template rule covers the document root and element nodes:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

This rule causes the XSL processor to process the immediate children of the document root or element, and their children as well if no successful pattern match is found. There is also another built-in template rule for each mode used in the template, as shown in the following:

```
<xsl:template match="*|/" mode="m">
  <xsl:apply-templates mode="m"/>
</xsl:template>
```

Just like the previous built-in rule, this template rule allows the XSL processor to recursively process the document root and element nodes in the absence of a successful pattern match.

The built-in template rule for text and attribute nodes, as shown in the following example, copies the text of each node to the result tree:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

The built-in template rule for processing instructions and comments is to do nothing, as shown in the following example:

```
<xsl:template match="processing-instruction()|comment()"/>
```

See also: xsl:apply-imports, xsl:apply-templates, xsl:call-template, xsl:import, xsl:include

xsl:text

```
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>
```

Outputs literal text to the result tree.

This element is defined in Section 7.2 of the XSLT 1.0 specification.

The body of a template may contain text node as well as literal result elements and elements within the XSL namespace. Text nodes are copied to the result tree. However, whitespace within the text nodes is normalized. There are situations where whitespace must be preserved. In those cases, wrap the text with an xsl:text element. The following example illustrates such a situation:

```
<xsl:template match="author">
  <xsl:text>Author: </xsl:text>
  <xsl:value-of select="."/>
</xsl:template>
```

The result tree is to contain the string “Author:” followed by the name of the author with a space separating the two. Without the xsl:text element, the space would not be copied to the result tree.

The xsl:text element may also be used to encapsulate a new line character, as shown in the following example:

```
<xsl:template match="author">
  <xsl:text>Author: </xsl:text>
  <xsl:value-of select="."/>
  <xsl:text>
  Born: </xsl:text>
  <xsl:value-of select="@birthdate"/>
</xsl:template>
```

The line feed between the start tag of the xsl:text element and the string “Born:” is retained by the XSL processor and copied to the result tree.

By default, the text within the xsl:text element is not escaped. To enabled output escaping, set the disable-output-escaping attribute to the value “no”.

See also: xsl:preserve-space, xsl:strip-space

xsl:transform

```
<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:transform>
```

Represents and declares a stylesheet.

This element is defined in Section 2.2 of the XSLT 1.0 specification and serves as a synonym for xsl:stylesheet. The xsl:transform attributes and usage are identical to xsl:stylesheet. See page 343 describing the xsl:stylesheet element earlier in this chapter for more information.

See also: xsl:import, xsl:stylesheet

xsl:value-of

```
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />
```

Creates a text node in the result tree based upon an expression.

This element is defined in Section 7.6.1 of the XSLT 1.0 specification.

Use this element to create a text node in the result tree. The select attribute is required and identifies an expression used to generate the contents of the text node. After the expression is evaluated, the XSL processor converts the result to a string. In the following template, the xsl:value-of element is used to output the value of a variable to the result tree:

```
  <xsl:value-of select="$displaytype"/>
```

The next example copies the value of an element's model attribute to the result tree:

```
<xsl:value-of select="@model"/>
```

The disable-output-escaping attribute is optional. It defaults to the value "yes" which means that characters are output as is to the result tree. To have the XSL processor escape special characters, set the value of this attribute to "no."

See also: xsl:copy-of

xsl:variable

```
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>
```

Declares a local or global variable.

This element is defined in Section 11 of the XSLT 1.0 specification.

Use this element to declare a variable that is global to the stylesheet or local to a template body. This element may appear as a top-level element or within a template body. The name attribute is required and serves to identify the variable.

The value of the variable may be calculated in two ways. First, the select attribute may be used to specify an expression that is converted into a string. Second, the body of the variable may be evaluated to determine the value. If the select attribute is specified then the body must be empty.

The following example declares a variable where the select attribute contains a literal string:

```
<xsl:variable name="Vendor" select="'Manifold West'"/>
```

The same variable could be declared with the literal string in the template body, as shown in the following declaration:

```
<xsl:variable name="Vendor">Manifold West</xsl:variable>
```

A variable may contain not only strings and numbers, but a node set. The following declaration selects all unshipped purchaseorder elements within the XML document:

```
<xsl:variable name="UnshippedPOs"
              select="//purchaseorder[@status='Pending']">
```

The body of the variable may contain decision-making logic as shown in the following example:

```
<xsl:variable name="color">
  <xsl:choose>
    <xsl:when test="@securitylevel = 'top-secret'">
      red
    </xsl:when>
    <xsl:otherwise>
      white
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

If the current node has a top secret security level then the variable color is set to the value "red"; otherwise, it defaults to "white."

A global variable is accessible throughout the entire stylesheet, including before it is declared and within any stylesheets that are included or imported. A local variable is accessible from the end tag of the local variable element to the end tag of the variable's parent element. The variable may not be accessed within its own template body.

Two local variables may share the same name as long as one is not declared within the other. A global variable may be redeclared as a local variable, overriding the original value.

See also: xsl:param

xsl:when

```
<xsl:when
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>
```

Defines a condition to be tested within an xsl:choose element.

This element is defined in Section 9.2 of the XSLT 1.0 specification.

The xsl:choose element provides the same functionality as a CASE or SWITCH statement in other programming languages. The xsl:choose element may contain multiple xsl:when elements and an optional xsl:otherwise element.

The test attribute is required and identifies a boolean condition that is to be tested. If the condition evaluates to True then the xsl:when template body is instantiated by the XSL processor. The template body may be empty. In either case, the XSL processor continues execution with the next node after the xsl:choose element.

The following snippet declares a variable named color whose value depends upon the securitylevel attribute of the current node:

```
<xsl:variable name="color">
  <xsl:choose>
    <xsl:when test="@securitylevel = 'top-secret'">
      red
    </xsl:when>
    <xsl:when test="@securitylevel = 'secret'">
      blue
    </xsl:when>
    <xsl:otherwise>
      white
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

See also: xsl:choose, xsl:if, xsl:otherwise

xsl:with-param

```
<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

Identifies a parameter passed to a named template.

This element is defined in Section 11.6 of the XSLT 1.0 specification.

Use this element to pass a parameter to a named template. A named template may accept zero or parameters, each having a unique name. When the template is invoked via the xsl:call-template element, use zero or more xsl:with-param elements to specify values for the template's parameters. The name attribute identifies the parameter. If a parameter of the same name does not exist in the named template then the parameter is ignored.

Set the value of the parameter using the select attribute or the body of the xsl:with-param element. If a select attribute is specified then the template body must be empty.

The purpose of the named template shown in the following example is to add two numbers and display their result:

```
<xsl:template name="add">
  <xsl:param name="A"/>
  <xsl:param name="B"/>
  <xsl:value-of select="$A"/>
  <xsl:text> + </xsl:text>
  <xsl:value-of select="$B"/>
  <xsl:text> = </xsl:text>
  <xsl:value-of select="$A + $B"/>
</xsl:template>
```

The template accepts two numbers, the first in parameter A and the second in parameter B. To add two numbers, the template would be called as follows:

```
<xsl:call-template name="add">
  <xsl:with-param name="A" select="1"/>
  <xsl:with-param name="B" select="1"/>
</xsl:call-template>
```

The string result of calling the template would be “1 + 1 = 2”.

See also: xsl:call-template, xsl:param, xsl:template

Chapter 9: XSL-Formatting Objects and Properties

This chapter provides an overview of XSL-Formatting Objects (XSL-FO) and provides a reference for the rich subset of formatting objects and properties supported by XMLPartner Professional's XSL processor. The first portion of this chapter explains formatting objects and properties and how to use them in an XSL stylesheet. The second portion of the chapter is a reference section for each of the formatting objects and properties supported by XMLPartner Professional.

XSL *formatting objects* are instructions to prepare an XML document for presentation as it is transformed. Chapter 8 describes how to transform an XML document into a new set of data using XPath and XSLT, but that type of transformation just creates a new set of data called a *result tree*. A document that is transformed using XPath and XSLT may be presentable, but is usually not ready for presentation. XSL formatting objects extend XPath and XSLT to allow creating a new set of data that is ready for presentation.

To be more specific, a printed report requires some aesthetic properties that are not included with the XML document and not provided by XSLT. For instance, a report needs a title page. Each page of the report may need a page number. Some of the data may need to be converted into a bulleted list or a table. Graphics may need to be added. Some of the text may need to be colorized. The formatting portion of XSL, formatting objects, handles these presentation tasks as well as many others.

Formatting objects are defined as elements in an XSL stylesheet. Each formatting object has options and may have required parameters. These options and parameters are called formatting properties and are defined as attributes of the formatting objects. Each of the formatting objects is described with a list of its associated formatting properties in the Formatting objects portion of the Reference Section on page 367.

The following example shows a portion of a stylesheet that displays a list of company names in a specific font for presentation:

```
<xsl:template match="COMPANY">
  <fo:block font-size="16pt" font-family="serif">
    <xsl:value-of select="@NAME"/>
  <fo:block/>
</xsl:template>
```

The previous example demonstrates that the fo:block formatting object is an element name that has formatting properties, called font-size and font-family, that are attributes of the element. Like other elements, the formatting objects may also have children. The fo:block object in the example contains an XSLT command that provides the text being formatted.

XSL formatting properties provide the details for implementing the XSL formatting object instructions. Some of the properties contain required information, such as telling a fo:external-graphic what file to retrieve. Other properties, like the ones in the previous example, are used to set options for a formatting object, such as printing text using a specific font or color. Each of the formatting properties is described in the Formatting properties portion of the Reference Section on page 392.

The fo:block and other formatting objects are described in the Reference Section later in this chapter. Before looking at the specific formatting objects it is important to understand the XSL area model and how pages in the new document are laid out.

XSL Area Model

The XSL area model is an ordered tree that describes the geometric structuring of the transformed document. Like all tree data structures, the base of the tree is called the *root*. As the XSL processor encounters formatting objects, it creates nodes in the tree called *areas*. Areas are rectangular objects containing data produced by the formatting objects. Areas can include lines, paragraphs, tables, and graphics among others. These areas are combined to form the transformed and formatted document.

There are two types of areas: *block* and *inline areas*. Inline areas are sections that flow from left to right and wrap around as lines fill up, like text. Block areas are sections that stack on top of each other, like paragraphs. Inline areas are contained inside of block areas. Areas are usually positioned in the new document in the same order as they're formed in the area tree, but some can be explicitly placed using formatting properties.

Formatting objects that create block areas include fo:block, fo:list-block, and fo:table. Formatting objects that create inline areas include fo:external-graphic, fo:leader, and fo:page-number. Inline areas can be turned into block areas by placing them in a fo:block object.

Page Sequences and Layout

When developing an XSL stylesheet that includes formatting objects, the first task is to design a page layout for each type of page. For instance, different page styles may be required for a title page, beginning of a chapter, and general content. The page layout specifies how pages are subdivided, how content is placed on pages, and any type of static data placed on each type of page.

As hinted at in the previous XSL area model section, the root of the formatting object tree must be a fo:root object. The fo:root must contain one fo:layout-master-set and a sequence of one or more fo:page-sequences as child nodes. The fo:layout-master-set object contains page-master (fo:simple-page-master or fo:page-sequence-master) children that determine the page layout and pagination. Content for the pages is determined by fo:page-sequence children.

The pages are comprised of *flows*, fo:flow and fo:static-text objects. fo:flow objects hold the bulk of the content that is distributed across a sequence of pages. fo:static-text objects hold data that is usually repeated on many pages and is retrieved in a static way, like a page number or chapter title. It is important to understand the content of a fo:static-text is not static (page numbers and chapter titles do change). It's the method of retrieving the content that is static. See the following section for an example.

Designing a layout

Each type of page used in the new document must be designed using a fo:simple-page-master object. The page can be divided into up to five regions: region-body, region-before, region-end, region-start, and region-after. The region-body, fo:region-body, is the center of the page and normally contains the bulk of the content. The region-before, fo:region-before, is located across the top of the page and normally serves as a header. The region-after, fo:region-after, is located across the bottom of the page and normally serves as a footer. The region-start, fo:region-start, is located on the left edge of the page and may be used as a left sidebar. The region-end, fo:region-end, is located on the right edge of the page and may be used as a right sidebar.

Each page can also be viewed in terms of *viewports*. A viewport is a child of the area tree that is viewable on the output medium. The page contains a *page-viewpoint-area* that can be divided into up to five regions called *region-viewport-areas*. The five region-viewport areas correspond with the fives regions described above.

Figure 9.1 is a visual representation of a page along with its regions and viewports.

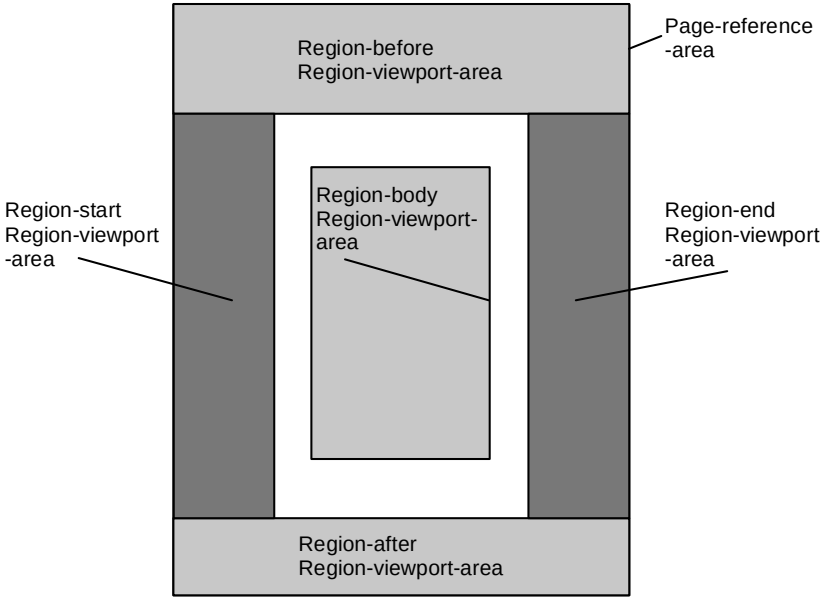


Figure 9.1: Visual representation of page and its regions and viewports (W3C, 2000)

Designing a sequence of pages

fo:page-sequence-master objects are used to identify a sequence of pages. The pages in the sequence refer to fo:simple-page-masters that have been defined. The page-sequence-master may contain a sequence of fo:single-page-master-reference objects, or it can specify a repetition of pages using a fo:repeatable-page-master-reference or multiple fo:repeatable-page-master-alternatives.

Use a single fo:single-page-master-reference object to instantiate a single page of output, like the title page of a report. Multiple fo:single-page-master-references can be used to explicitly instantiate a sequence of individual pages. Note that it is an error if all the flow objects generated by the fo:page-sequence do not fit on the single-page-master-references. The amount of content is not a concern for interactive documents, like HTML, whose page size is dynamic.

Use fo:repeatable-page-master-references to account for a varying number of pages. fo:repeatable-page-master-references instantiate enough pages to contain all the flow objects generated by the fo:flow-sequence.

Pages can also be instantiated conditionally using fo:repeatable-page-master-alternatives. fo:repeatable-page-master-alternatives instantiate pages based on a set of conditions. The conditions include page position (first, last, etc.), whether the page is blank, and whether the page is odd or even. This allows for such things as defining a separate page type for even and odd pages and having them instantiated as needed. Use a fo:conditional-page-master-reference to setup the conditions.

The following example is a portion of an XSL stylesheet that brings all this together to identify a 8 ½ x 11 inch page that is filled with the results of the fo:flow object's call to xsl:apply-templates and static text in the footer containing "Page " followed by the current page number.

```
<fo:root>
   <fo:layout-master-set>
     <fo:simple-page-master master-name="page"
       page-height="11in" page-width="8.5in"
       margin-top="0.5in" margin-bottom="0.5in"
       margin-left="0.5in" margin-right="0.5in">
       <fo:region-body
         margin-top="0in" margin-bottom="0.5in"
         margin-left="0in" margin-right="0in"/>
       <fo:region-after/>
     </fo:simple-page-master>
   </fo:layout-master-set>
   <fo:page-sequence master-name="content">
     <fo:static-content flow-name="xsl-region-after">
       <fo:block>
         <xsl:text>Page </xsl:text>
         <fo:page-number/>
       </fo:block>
     </fo:static-content>
     <fo:flow flow-name="xsl-region-body">
       <xsl:apply-templates/>
     </fo:flow>
   </fo:page-sequence>
 </fo:root>
```

Putting the pages together

Figure 9.2 helps visualize how the page layout and sequence objects fit together.

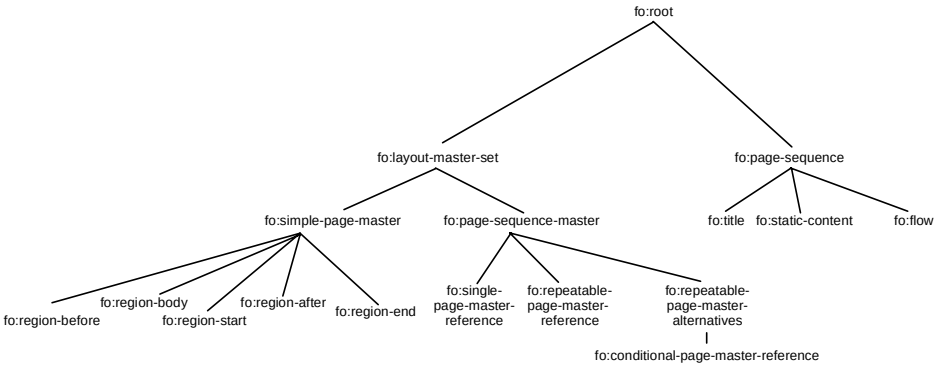


Figure 9.2: Tree representation of the formatting objects for pagination (W3C, 2000)

9

Formatting Content

After designing the page layout and creating a sequence of pages, the next step is to format the content of the new document. There are many options for formatting content. Text can be made bold, italicized, colorized, etc. Groups of text can be turned into bulleted lists or tables. Graphics, rules, links, and leaders can be added. The following sections discuss how to accomplish many of these formatting tasks.

Text

Text for paragraphs, titles, captions, and headlines is normally grouped and formatted as a block. A block is formed using the fo:block formatting object. fo:block creates a block area and has properties to control the formatting of text, borders, breaks before and after blocks, margins, and hyphenation.

To format inline text, use the fo:inline object. This formatting object is useful for making a word bold or italicized within a sentence.

The following example creates a block of text containing the word “Page” followed by the current page number. The fo:block properties sets the font to 12-point Courier. The fo:inline property makes the page number bold.


```
<fo:block font-family="Courier"
          font-size="12pt">
  <xsl:text>Page </xsl:text>
  <fo:inline font-weight="bold">
    <fo:page-number/>
  </fo:inline>
</fo:block>
```

Lists

Lists are used to associate a contiguous series of items together, like steps in a process or a list of things to accomplish. Lists are generated using the fo:list-block, fo:list-item, fo:list-item-label, and fo:list-item-body objects. fo:list-block is a block-level object that holds the items in the list. Each of the items is contained in a fo:list-item object which contains a fo:list-item-label and fo:list-item-body. The fo:list-item-label contains a label for the list item. The label can be a number, symbol, graphic, or virtually anything that might be used to identify a list item. The fo:list-item-body contains the content of the list-item. Like the label, the body can contain virtually any type of data.

The following XML document contains a grocery list:

```
<gl>
  <item>milk</item>
  <item>eggs</item>
  <item>cheese</item>
</gl>
```

The following portion of an XSL stylesheet would create a bulleted list:

```
<xsl:template match="gl">
  <fo:list-block>
    <xsl:apply-templates/>
  </fo:list-block>
</xsl:template>

<xsl:template match="gl/item">
  <fo:list-item>
    <fo:list-item-label>
      <fo:block>
        <xsl:number format="a."/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>
        <xsl:apply-templates/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
```

This transformation would produce the following list:

a. milk

b. eggs

c. cheese

Tables

A table is a collection of information that is presented in columns and rows. Tables are generated using the fo:table, fo:table-body, fo:table-column, fo:table-row, and fo:table-cell objects. fo:table is a block-level object that defines the table's columns and body. Each column must be defined using fo:table-column objects. fo:table-body contains the rows, fo:table-rows, or cells, fo:table-cells, that make up the content of the table.

The following XSL stylesheet is an example of a table showing a list of colors and their associated hexadecimal RGB value. The table has two columns and three rows. The first column contains the name of a color and the second row contains the color's RGB value. The cells in the top row are used as headers for the columns and the other two rows contain entries for two colors.

```
<fo:table>
<fo:table-body>
    <fo:table-column column-number="1"
                     column-width="144pt">
    </fo:table-column>
    <fo:table-column column-number="2"
                     column-width="144pt">
    </fo:table-column>
    <fo:table-row>
      <fo:table-cell>
        <fo:block font-family="Courier, monospace"
                  font-weight=”bold”>
          Color
        </fo:block>
      </fo:table-cell>
      <fo:table-cell>
        <fo:block font-family="Courier, monospace"
                  font-weight="bold">
          Hexadecimal RGB Value
        </fo:block>
      </fo:table-cell>
    </fo:table-row>
    <fo:table-row>
      <fo:table-cell>
        <fo:block font-family="Courier, monospace">
          chartreuse
        </fo:block>
      </fo:table-cell>
      <fo:table-cell>
        <fo:block font-family="Courier, monospace">
          #7FFF00
```

```
          </fo:block>
        </fo:table-cell>
      </fo:table-row>
      <fo:table-row>
        <fo:table-cell>
          <fo:block>hotpink</fo:block>
        </fo:table-cell>
        <fo:table-cell>
          <fo:block font-family="Courier, monospace">
            #FF69F4
          </fo:block>
        </fo:table-cell>
      </fo:table-row>
    </fo:table-body>
  </fo:table>
```

This stylesheet produces the table shown in Figure 9.3.

Color	Hexadecimal RGB Value
chartreuse	#7FFF00
hotpink	#FF69F4

Figure 9.3: Stylesheet table example

Leaders and rules

A leader is a series of dots, dashes, or other text used to direct the eye across a page. A leader is typically used in a book's table of contents to direct the reader's eye from a subject to a page number. A rule is a line that separates elements on a page. Leaders and rules are created using the fo:leader object. The fo:leader's leader-pattern property determines how the leader is drawn; a pattern for a leader or a solid line for a rule.

The following stylesheet creates a rule that is five inches long and 1 point wide:

```
<fo:leader leader-length="5in" leader-pattern="rule"
           rule-thickness="1pt"/>
```

Graphics

Graphics are retrieved from an external source using the fo:external-graphic object. fo:external-graphic is a inline-level object. Like all inline areas, it can be turned into a block-level object by making it a child of a fo:block object. The object has no children. Its src property points to a URL containing the graphic to be retrieved.

The following portion of an XSL stylesheet retrieves a graphic who's location is stored as an attribute named "photo":

```
<xsl:template match="house">
  <fo:block>
    <fo:external-graphic src="{@photo}"/>
  </fo:block>
</xsl:template>
```

Links

Links are objects that act like HTML hyperlinks. They are used to jump between documents or places within a document. This doesn't have any effect for non-interactive mediums, but is a valuable resource for interactive mediums like HTML. Hyperlinks are created using the fo:basic-link object. The object's internal-destination property is used to jump to a new place in the current document. To jump to a spot in another document, use the external-destination property as shown in the following example:

```
<fo:block>Be sure to visit
  <fo:basic-link external-
destination="http://www.turbopower.com"/>
    TurboPower Software
  </fo:basic-link>
</fo:block>
```

Reference

This Reference Section is divided into two sections: Formatting objects and Formatting properties. Each formatting object contains a purpose, a description, a list of associated properties, and a list of related formatting properties. The purpose is the short description prefixed with an arrow ↳and the longer description follows the purpose. The applicable formatting properties are listed below the description. The last portion is the list of associated formatting objects which is prefixed with "See also:."

The Formatting properties section is very similar, but the list of applicable formatting properties is replaced with a list of formatting objects to which the formatting property applies. Each formatting property also includes its formatting property tag code listed on the right side of the property name.

Formatting objects

basic-link

`fo:basic-link`

↳ Defines a link to a location within the current document or to an external resource.

fo:basic-link generates an inline area that jumps to a target location within the same document or to another document. This object requires at least one inline- or block-level child to form the link to the target. The object must have either an external-destination or an internal-destination property. The show-destination property determines if the target replaces the current document or if it appears in a new window. The indicate-destination property specifies if the system should show what the target is.

Note that this object only applies to interactive media such as HTML.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width

external-destination	padding-bottom
id	padding-end
keep-together	padding-left
keep-with-next	padding-right
keep-with-previous	padding-start
line-height	padding-top
line-height-shift-adjustment	show-destination
padding-after	space-end
padding-before	space-start

block

`fo:block`

Groups a section of related content.

This command is commonly used to format paragraphs, titles, headlines, figures, table captions, and other sections of related content. It usually specifies a rectangular area that occupies the width of the containing area and a height that is determined by the amount of content that the block contains. A block may specify separation between it and a preceding or subsequent block-level object as well as unique indents on the start of the first text line of the block and end of the last text line of the block.

The following formatting properties apply to this object:

background-attachment	border-left-width
background-color	border-right-color
background-image	border-right-width
background-position-horizontal	border-start-color
background-position-vertical	border-start-width
background-repeat	border-top-color
border-after-color	border-top-width
border-after-width	break-after
border-before-color	break-before
border-before-width	color
border-bottom-color	end-indent
border-bottom-width	font-family
border-end-color	font-size
border-end-width	font-size-adjust
border-left-color	font-stretch

font-style
font-variant
font-weight
hyphenate
hyphenation-character
hyphenation-keep
hyphenation-push-charter-count
hyphenation-remain-character-count
id
keep-together
keep-with-next
keep-with-previous
language
line-height
line-height-shift-adjustment
margin-bottom
margin-left
margin-right
margin-top
orphans
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
space-after
space-before
start-indent
text-align
text-align-last
text-indent
widows
wrap-option

conditional-page-master-reference

`fo:conditional-page-master-reference`

Specifies the conditions for a repeating, alternating page sequence.

fo:conditional-page-master-reference encompasses three conditions: blank-or-not-blank, odd-or-even, and page-position. The conditions are specified using the object's corresponding properties. All three conditions must be satisfied before the page master is selected. The master-name property must be set to the value of the simple-page-master to use when the conditions are satisfied.

The following formatting properties apply to this object:

blank-or-not-blank
master-name
odd-or-even
page-position

See also: repeatable-page-master-alternatives

`fo:external-graphic`

Retrieves a graphic that resides outside of the result tree.

The graphic is rendered on the output medium in its original size if the content-height and content-width properties are not specified. The src property is required to identify the source of the graphic.

This object returns an inline area (viewport) and does not contain any children. The height and width properties determine the size of the inline area that is returned. The content-height and content-width determine the size of the graphic in the viewport.

The following formatting properties apply to this object:

alignment-adjust
background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
content-height
content-type
content-width
height
id
keep-with-next
keep-with-previous
line-height
line-height-shift-adjustment
overflow
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
scaling
scaling-method
space-end
space-start
src
text-align
width

flow

`fo:flow`

Defines a set of flow objects that provide content for a sequence of pages.

This object does not return any objects. It only encapsulates the flow objects contained in its block-level children. A flow object is a formatting object that either generates an area or returns areas. This object must contain at least one block-level area child.

The following formatting property applies to this object:

flow-name

See also: static-content

inline

`fo:inline`

Formats inline content.

fo:inline is often used style a portion of inline text. The style can include formatting text, adding a background, and adding a border around the content. This object differs from fo:block because of the type of area that is created: fo:block creates a block area and fo:inline creates an inline area. This object can contain text, inline areas, or block areas.

The following formatting properties apply to this object:

alignment-adjust
background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
color
font-family
font-size
font-size-adjust
font-stretch
font-style
font-variant
font-weight
id

keep-together
keep-with-next
keep-with-previous
line-height
line-height-shift-adjustment
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
space-end
space-start

layout-master-set

`fo:layout-master-set`

Defines a set of page-layout and page-sequence definitions.

Page layouts are defined using simple-page-master objects and page sequences are defined using page-sequence-masters. These masters provide a set of templates to hold the content generated by the root's page-sequence children.

This object requires at least one simple-page-master and at least one page-sequence-master object. This object does not have any properties.

See "Page Sequences and Layout" on page 358 for more information about this object.

See also: page-sequence, page-sequence-master, root, simple-page-master

leader

`fo:leader`

Creates a leader or a rule.

A leader is a set of dots, dashes, or other content that is used to direct a reader's eyes across a page. A leader is typically used in a book's table of contents to direct a reader's eyes from a subject to its page number. A rule is a line that is used to separate portions of text, like the one under each entry in this reference section.

The leader-pattern property determines how the leader is drawn. Set the leader-pattern to "rule" to create a solid line. A leader pattern value of "dots" creates a typical leader of dots and a value of "use-content" creates a leader using a repeated pattern of the leader object's children. See the leader-pattern property in the next portion of the reference section for more information.

This object returns an inline area. It can contain any number of inline area children, except fo:leaders. These children are ignored unless the leader-pattern property is set to “use-content.” If the leader-pattern property is set to “use-content” and the leader doesn’t have any children, the leader is created as a blank line.

The following formatting properties apply to this object:

alignment-adjust
background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
color
font-family
font-size
font-size-adjust
font-stretch
font-style
font-variant
font-weight
id
leader-length
leader-pattern
letter-spacing
line-height
line-height-shift-adjustment
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
rule-thickness
space-end
space-start
word-spacing

`fo:list-block`

Defines a group of items contained in a list.

The fo:list-block object generates a block-level area that contains a list of items to place in a list. The items in the list are defined using the fo:list-item object. The list-block must contain at least one list-item child. fo:list-item objects are the only valid children of this object.

See “Formatting Content” on page 362 for additional information and an example.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- break-after
- break-before
- end-indent
- id
- keep-together
- keep-with-next
- keep-with-previous
- margin-bottom
- margin-left
- margin-right
- margin-top
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- provisional-distance-between-starts
- provisional-label-separation
- space-after
- space-before
- start-indent

See also: list-body, list-item, list-label

list-item

`fo:list-item`

Contains a list label and body that form a list item.

A label is an identifier for each item in the list and is defined using a fo:list-label object. Typical labels include bullets or numbers. The body is the content of the list item. Each list-item object must contain one list-item-body and one list-item-label.

See “Formatting Content” on page 362 for additional information and an example.

The following formatting properties apply to this object:

background-attachment	break-after
background-color	break-before
background-image	end-indent
background-position-horizontal	id
background-position-vertical	keep-together
background-repeat	keep-with-next
border-after-color	keep-with-previous
border-after-width	margin-bottom
border-before-color	margin-left
border-before-width	margin-right
border-bottom-color	margin-top
border-bottom-width	padding-after
border-end-color	padding-before
border-end-width	padding-bottom
border-left-color	padding-end
border-left-width	padding-left
border-right-color	padding-right
border-right-width	padding-start
border-start-color	padding-top
border-start-width	space-after
border-top-color	space-before
border-top-width	start-indent

See also: list-block, list-item, list-item-body, list-item-label

list-item-body

`fo:list-item-body`

Defines the content for a list item.

Content is formed by concatenating its block-level children. fo:list-item-body must have at least one block-level child.

See “Formatting Content” on page 362 for additional information and an example.

In the following example, “milk”, “eggs”, and “cheese” are the three list item bodies:

- milk
- eggs
- cheese

The following formatting properties apply to this object:

id	keep-together

See also: list-block, list-item, list-item-label

list-item-label

`fo:list-item-label`

Defines an enumeration, identifier, or decoration for a list item.

Labels often include numbers, bullets, or icons used to identify an individual list item. list-item-label objects must contain at least one block-level child.

See “Formatting Content” on page 362 for additional information and an example.

In the following example, “1. ”, “2. ”, “3. ”, and “4. ” are the four item labels:

1. Shake paint can vigorously.
2. Open paint can.
3. Insert paintbrush into paint can.
4. Move paintbrush across surface to be painted.

The following formatting properties apply to this object:

id	keep-together

See also: list-block, list-item, list-item-body

page-number

`fo:page-number`

Retrieves the page number of the current inline-area.

fo:page-number returns an inline area and is often used in fo:static-content to place page numbers in the header or footer of each page in the new document.

The following example shows a portion of an XSL stylesheet that defines a page with the current page number in the footer:

```
<fo:page-sequence master-name="content">
  <fo:static-content flow-name="xsl-region-after">
    <fo:block>
      <xsl:text>Page </xsl:text>
      <fo:page-number/>
    </fo:block>
  </fo:static-content>
  <fo:flow flow-name="xsl-region-body">
    <xsl:apply-templates/>
  </fo:flow>
</fo:page-sequence>
```

The following formatting properties apply to this object:

alignment-adjust	border-end-color
background-attachment	border-end-width
background-color	border-left-color
background-image	border-left-width
background-position-horizontal	border-right-color
background-position-vertical	border-right-width
background-repeat	border-start-color
border-after-color	border-start-width
border-after-width	border-top-color
border-before-color	border-top-width
border-before-width	font-family
border-bottom-color	font-size
border-bottom-width	font-size-adjust

font-stretch
font-style
font-variant
font-weight
id
keep-with-next
keep-with-previous
letter-spacing
line-height
line-height-shift-adjustment
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
score-spaces
space-end
space-start
word-spacing

page-sequence

`fo:page-sequence`

Defines the content for a page or sequence of pages.

fo:page-sequences are used to generate text that is placed in the templates defined within the layout-master-set object.

9

page-sequence objects contain: one flow object, an optional title objects, and may contain many static-content objects, but none are required. The master-name property is required to select the page master that is used create the pages used by the content of this sequence.

See “Page Sequences and Layout” on page 358 for more information and an example.

The following formatting properties apply to this object:

id
master-name

See also: flow, layout-master-set, root, static-content, title

page-sequence-master

`fo:page-sequence-master`

Specifies pages or sequences of pages that contain the transformed document's content.

Page masters define the geometry and layout of an individual page. The page-sequence-master determines when the page masters are used. There are three types of sequences defined by the page-sequence-master: an individual page, a repeatable page, and a conditionally repeated page. The sequence types are defined using single-page-master-reference, repeatable-page-master-reference, and repeatable-page-master-alternatives children respectively.

Note that it is an error to exhaust all page-master-references before all content (flow objects) have been place on a page. For instance, if the page-sequence-master only defines one single-page-master-reference and the assigned page-sequence produces more than one page, it is an error. Use the repeatable reference types to prevent this type of error.

The page-sequence-master must define at least one page-master-reference and the master-name property must be assigned the name page-master that is defined within the same layout-master-set object.

See "Page Sequences and Layout" on page 358 for more information and an example.

The following formatting property applies to this object:

master-name

See also: layout-master-set, repeatable-page-master-alternative, repeatable-page-master-reference, single-page-master-reference

region-after

`fo:region-after`

Adds a footer section to a page.

This object adds an optional footer to a simple-page-master. See “Page Sequences and Layout” on page 358 for a description of page layout.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- display-align
- overflow
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- region-name

See also: simple-page-master

region-before

`fo:region-before`

Adds a header to page.

This object adds an optional header to a simple-page-master. See “Page Sequences and Layout” on page 358 for a description of page layout.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- display-align
- overflow
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- region-name

See also: simple-page-master

region-body

`fo:region-body`

Adds the main content section to a page.

This object adds the required content section to a simple-page-master. The region-body is the area in the center of the page. See “Page Sequences and Layout” on page 358 for a description of page layout.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- display-align
- end-indent
- margin-bottom
- margin-left
- margin-right
- margin-top
- overflow
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- region-name
- space-after
- space-before
- start-indent

See also: simple-page-master

region-end

`fo:region-end`

Adds a right sidebar to a page.

This object adds an optional right sidebar to a simple-page-master. See “Page Sequences and Layout” on page 358 for a description of page layout.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- display-align
- overflow
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- region-name

See also: simple-page-master

region-start

`fo:region-start`

Adds an optional left sidebar to a page.

This object adds an optional left sidebar to a simple-page-master. See “Page Sequences and Layout” on page 358 for a description of page layout.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- display-align
- overflow
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- region-name

See also: simple-page-master

repeatable-page-master-alternatives

`fo:repeatable-page-master-alternatives`

Specifies a conditionally repeated page.

Include repeatable-page-master-alternative objects in a page-sequence-master when a page, simple-page-master, is used repetitively based on a condition. Conditions include page position (first, last, etc.), odd or even page number, or if the page is blank. This object is required to have one or more conditional-page-master-reference children to establish its conditions. The master-name property is also required to identify the simple-page-master.

If the page is to be repeated without any conditions, use repeatable-page-master-reference instead.

See also: conditional-page-master-reference, page-sequence, page-sequence-master, repeatable-page-master-alternatives, simple-page-master, single-page-master-reference

repeatable-page-master-reference

`fo:repeatable-page-master-reference`

Defines a repeated sequence of pages.

Include repeatable-page-master-references objects in a page-sequence-master when the page, simple-page-master, is used repetitively. The master-name property is required to identify the simple-page-master.

If the page is to be used conditionally, use repeatable-page-master-alternatives instead.

The following formatting property applies to this object:

master-name

See also: page-sequence, page-sequence-master, repeatable-page-master-alternatives, simple-page-master, single-page-master-reference

root

`fo:root`

Defines the top node of the formatting object tree.

This is the top node of the formatting object tree. It is simply a container for a layout-master-set and one or more page-sequences.

This object does not have any formatting properties.

See also: layout-master-set, page-sequence

simple-page-master

`fo:simple-page-master`

Defines the layout of a page.

Each type of page used in a presentation must have its layout defined using a simple-page-master. The simple-page-master's page-height and page-width properties define the size of the page. The object's region children define the subsections of the page.

The object must have one region-body child and may have an instance of region-after, region-before, region-start, and region-end. The simple-page-master must have a unique name assigned to its master-name property.

The following formatting properties apply to this object:

end-indent	page-height
margin-bottom	page-width
margin-left	space-after
margin-right	space-before
margin-top	start-indent
master-name	

See also: conditional-page-master-reference, layout-master-set, page-sequence, page-sequence-master, region-after, region-before, region-body, region-end, region-start

single-page-master-reference

`fo:single-page-master-reference`

Specifies a single-page page sequence.

Use fo:single-page-master-reference to specify that an individual simple-page-master is used to hold a portion of content from the transformed document. Note that it is an error if the generated content will not fit on the single page.

This object does not have any children. The master-name property is required to specify the simple-page-master.

The following formatting property applies to this object:

master-name

See also: page-sequence, repeatable-page-master-alternative, repeatable-page-master-reference, simple-page-master

static-content

`fo:static-content`

Defines static content on one or more pages.

Use fo:static-content to place static content on a page or sequence of pages. The content can be a page number, the title of a report, a graphic, or any data that is calculated in static way.

This object must have one or more block children containing the static content and the flow-name property must be assigned a name that is unique within its page sequence.

The following formatting property applies to this object:

flow-name

See also: page-sequence

table

`fo:table`

Formats content within a table.

fo:table describes the columns, rows, and cells that make up a table. Columns are described using one or more fo:table-column objects and the rows and cells are described within one or more fo:table-body objects.

See “Formatting Content” on page 362 for additional information and an example of creating a table.

The following formatting properties apply to this object:

background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
break-after
break-before
end-indent
height

id
keep-together
keep-with-next
keep-with-previous
margin-bottom
margin-left
margin-right
margin-top
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
space-after
space-before
start-indent
width

See also: table-body, table-cell, table-column, table-row

table-body

`fo:table-body`

Defines the rows contained in the body of a table.

fo:table-body requires one or more table-row children or one or more table-cell children. If table-cell children are used, their starts-row and ends-row properties must be used to define each row.

See “Formatting Content” on page 362 for additional information and an example of creating a table.

The following formatting properties apply to this object:

background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
id
padding-after

padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top

See also: table, table-cell, table-row

table-cell

`fo:table-cell`

Formats content placed in a table cell.

Content is retrieved from the object's block-level children.

The starts-row and ends-row properties must be used to define each row.

See "Formatting Content" on page 362 for additional information and an example of creating a table.

The following formatting properties apply to this object:

background-attachment
background-color
background-image
background-position-horizontal
background-position-vertical
background-repeat
border-after-color
border-after-width
border-before-color
border-before-width
border-bottom-color
border-bottom-width
border-end-color
border-end-width
border-left-color
border-left-width
border-right-color
border-right-width
border-start-color
border-start-width
border-top-color
border-top-width
column-number
display-align
empty-cells
ends-row
height
id
padding-after
padding-before
padding-bottom
padding-end
padding-left
padding-right
padding-start
padding-top
starts-row
width

See also: table-body, table-row

table-column

`fo:table-column`

Defines a column within a table.

fo:table-column defines the characteristics of a table's cells that have the same column and span. The most important characteristic is the column width, which is specified using the column-width property.

See "Formatting Content" on page 362 for additional information and an example of creating a table.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- column-number
- column-width
- number-columns-repeated

See also: table

table-row

`fo:table-row`

Groups table cells into rows.

Since each row must contain at least one cell, fo:table-row must contain one or more fo:table-cell children. fo:table-row's fo:table-cell children are grouped into a row.

See "Formatting Content" on page 362 for additional information and an example of creating a table.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- break-after
- break-before
- height
- id
- keep-together
- keep-with-next
- keep-with-previous

See also: table-body

title

`fo:title`

Defines a title for a document.

The use of the title object is implementation independent. In XMLPartner Professional, it is only used to create a title element of the HTML document created by the HTML filter.

The following formatting properties apply to this object:

- background-attachment
- background-color
- background-image
- background-position-horizontal
- background-position-vertical
- background-repeat
- border-after-color
- border-after-width
- border-before-color
- border-before-width
- border-bottom-color
- border-bottom-width
- border-end-color
- border-end-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-start-color
- border-start-width
- border-top-color
- border-top-width
- color
- font-family
- font-size
- font-size-adjust
- font-stretch
- font-style
- font-variant
- font-weight
- line-height
- line-height-shift-adjustment
- padding-after
- padding-before
- padding-bottom
- padding-end
- padding-left
- padding-right
- padding-start
- padding-top
- space-end
- space-start

alignment-adjust FOP _ALIGNMENT_ADJUST

`alignment-adjust`

Default: auto

Specifies an alignment point for inline areas.

This property allows more precise alignment of areas generated by formatting objects. It is particularly useful for graphics since they do not have a baseline.

The following table describes the possible values for this property:

Value	Meaning
auto	The alignment-point is at the intersection of the start-edge of the allocation-rectangle.
baseline	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the dominant-baseline of the area.
after-edge	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the after-edge baseline of the area.
central	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the central baseline of the area.
mathematical	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the mathematical baseline of the area.
middle	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the middle baseline of the area.
before-edge	The alignment-point is at the intersection of the start-edge of the allocation-rectangle and the before-edge baseline of the area.

This property applies to the following formatting objects:

external-graphic	leader
inline	page-number

`background-attachment`

Default: scroll

Specifies whether a background image is fixed to a viewport or scrolls with a document.

When the transformed document is rendered on interactive media, like HTML, this property determines if the background is fixed within the viewport. A viewport is a child area of the area root tree. This property has no meaning when the background-image property is not used.

The two options for this property are described in the following table:

Option	Description
fixed	The background-image is fixed within the viewable area of the enclosing object.
scroll	The background-image may scroll with the enclosing object.

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- table-column
- table-row
- title

See also: background-image

background-color FOP_BACKGROUND_COLOR

`background-color`

Default: transparent

Specifies an area's background color.

The property can have a value of "transparent" or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See "Predefined colors for formatting properties" on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-body
block	region-end
external-graphic	region-start
inline	table
leader	table-body
list-block	table-cell
list-item	table-column
page-number	table-row
region-after	title
region-before	

background-image FOP_BACKGROUND_IMAGE

`background-image`

Default: none

Specifies an image that is drawn on an area's background.

To have an image drawn on an area's background, assign this property a URI-specification to the location of an image. Supported image types are tiff, jpeg, png, and bmp. Assign a value of "none" to prevent drawing an image on the background.

This property applies to the following formatting objects:

basic-link	region-body
block	region-end
external-graphic	region-start
inline	table
leader	table-body
list-block	table-cell
list-item	table-column
page-number	table-row
region-after	title
region-before	

See also: background-attachment, background-position-horizontal, background-position-vertical

background-position-horizontal FOP_BACKGROUND_POSITION_H

`background-position-horizontal`

Default: 0pt

Specifies the initial horizontal position of a background image.

The following table describes the possible values for this property:

Value	Meaning
`left`	The left-edge of the image is aligned with the left-edge of the area's padding-rectangle.
`center`	Centers the image horizontally within the padding rectangle.
`right`	Places the right-edge of the image aligned with the right-edge of the padding-rectangle.
`<length>`	Specifies that the left-edge of the image shall be placed at the specified length to the right of the left-edge of the padding-rectangle.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- table-column
- table-row
- title

See also: background-image

background-position-vertical FOP_BACKGROUND_POSITION_V

`background-position-vertical`

Default: 0pt

Specifies the initial vertical position of a background image.

The following table describes the possible values for this property:

Value	Meaning
left	The top-edge of the image is aligned with the top-edge of the area's padding-rectangle.
center	Centers the image vertically within the padding rectangle.
right	Places the bottom-edge of the image aligned with the bottom-edge of the padding-rectangle.
<length>	Specifies that the top-edge of the image shall be placed at the specified length below the top-edge of the padding-rectangle.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- table-column
- table-row
- title

See also: background-image

`blank-or-not-blank`

Default: any

Defines a blank page condition for a conditional page sequence.

This possible values for this property are described in the following table:

Value	Meaning
`any`	This master is always eligible for selection.
`blank`	This master is eligible for selection if a page must be generated and there are no areas from the fo:flow to be put on that page.
`not-blank`	This master is eligible for selection if this page contains areas from the fo:flow.

This property applies to the following formatting object:

conditional-page-master-reference

border-after-color FOP_BORDER_AFTER _COLOR

`border-after-color`

Specifies a color for the border for the after edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link
block
external-graphic
inline
leader
list-block
list-item
page-number
region-after
region-before
region-body
region-end
region-start
table
table-body
table-cell
title

`border-after-width`

Default: medium

Specifies the width of the border's after edge.

The possible values for this property are shown in the following table:

Value	Meaning
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`border-before-color`

Specifies a color for the border for the before edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-before-width FOP_BORDER_BEFORE_WIDTH

`border-before-width`

Default: medium

Specifies the width of the border’s before edge.

The possible values for this property are shown in the following table:

Value	Meaning
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border’s thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

border-bottom-color FOP_BORDER_BOTTOM _COLOR

`border-bottom-color`

Specifies a color for the border for the bottom edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See the Appendix for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-bottom-width FOP_BORDER_BOTTOM_WIDTH

`border-bottom-width`

Default: medium

Specifies the width of the border's bottom edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-end-color FOP_BORDER_END_COLOR

`border-end-color`

Specifies a color for the border for the end edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

`border-end-width`

Default: medium

Specifies the width of the border's end edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`border-left-color`

Specifies a color for the border for the left edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-left-width FOP_BORDER_LEFT_WIDTH

`border-left-width`

Default: medium

Specifies the width of the border’s left edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border’s thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

border-right-color FOP_BORDER_RIGHT _COLOR

`border-right-color`

Specifies a color for the border for the right edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block

list-item	region-start
page-number	table
region-after	table-body
region-before	table-cell
region-body	title
region-end	

border-right-width FOP_BORDER_RIGHT_WIDTH

`border-right-width`

Default: medium

Specifies the width of the border's right edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-start-color FOP_BORDER_START_COLOR

`border-start-color`

Specifies a color for the border for the start edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

`border-start-width`

Default: medium

Specifies the width of the border's start edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`border-top-color`

Specifies a color for the border for the top edge of an area.

The property can have a value of “transparent” or a valid color specification. The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

border-top-width FOP_BORDER_TOP_WIDTH

`border-top-width`

Default: medium

Specifies the width of the border’s top edge.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border’s thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

break-after FOP_BREAK_AFTER

`break-after`

Default: auto

Forces the area generated by a formatting object to be the last area within a context.

The following table describes the possible values for this property:

Value	Meaning
`auto`	Breaks are not forced.
`column`	Imposes a break-after condition with a context consisting of column-areas.
`page`	Imposes a break-after condition with a context consisting of page-areas.
`even-page`	Imposes a break-after condition with a context consisting of even page-areas. A blank page is generated if necessary.
`odd-page`	Imposes a break-after condition with a context consisting of odd page-areas. A blank page is generated if necessary.

This property applies to the following formatting objects:

block
list-block
list-item
table
table-row

See also: break-before, keep-together, keep-with-next, keep-with-previous, orphan, widow

9

break-before FOP_BREAK_BEFORE

`break-before`

Default: auto

Forces the area generated by the current formatting object to be first within a context.

The following table describes the possible values for this property:

Value	Meaning
auto	Breaks are not forced.
Column	Imposes a break-before condition with a context consisting of column-areas.
Page	Imposes a break-before condition with a context consisting of page-areas.
even-page	Imposes a break-before condition with a context consisting of even page-areas. A blank page is generated if necessary.
odd-page	Imposes a break-before condition with a context consisting of odd page-areas. A blank page is generated if necessary.

This property applies to the following formatting objects:

block	table
list-block	table-row
list-item	

See also: break-after, keep-together, keep-with-next, keep-with-previous, orphan, widow

color FOP_COLOR

`color`

Specifies a color for the area returned by the formatting object.

The color specification can be a predefined name of a color a hexadecimal representation of its RGB values. See “Predefined colors for formatting properties” on page 468 for a list of predefined colors and their RGB values.

This property applies to the following formatting objects:

block	leader
inline	title

column-number FOP_COLUMN_NUMBER

`column-number`

Specifies a numerical identifier for a column.

For a table-column, the column-property assigns a unique numerical identifier for the column. The initial value is ‘1’ for the first column and the previous column number plus one for all other columns.

For a table-cell, the column-number property specifies the first column number spanned by the cell. The initial value is ‘1’ for the first cell defined in a row and the previous cell’s column-number of the previous cell plus the number of cells spanned by the previous cell.

This property applies to the following formatting objects:

table-cell table-column

See also: number-columns-repeated

column-width FOP_COLUMN_WIDTH

`column-width`

Specifies the width of table column.

This property must be set for every column. The length must be specified using one of the units from the following table:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

table-column

content-height FOP_CONTENT_HEIGHT

`content-height`

Default: auto

Specifies the height of the content within the viewport returned by the formatting object.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

external-graphic

See also: content-width

content-width FOP_CONTENT_WIDTH

`content-width`

Default: auto

Specifies the width of the content within the viewport returned by the formatting object.

The possible values for this property are shown in the following table:

Value	Meaing
thin	A thin border.
medium	A medium border.
thick	A thick border.
<length>	The border's thickness has an explicit value. Explicit border widths cannot be negative.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

external-graphic

See also: content-height

display-align FOP_DISPLAY_ALIGN

`display-align`

Default: auto

Specifies how child areas are aligned within a reference area.

The following table describes the possible values for this property:

Value	Meaning
auto	Same as before.
before	The top edge of the first child area is aligned with the top edge of the reference area.
after	The bottom edge of the last child is aligned with the bottom edge of the reference area.

This property applies to the following formatting objects:

region-after	region-end
region-before	region-start
region-body	table-cell

empty-cells FOP_EMPTY_CELLS

`empty-cells`

Default: show

Controls the borders of cells that have no visible content.

The possible values for this property are described in the following table:

Value	Meaning
show	Borders are drawn around empty cells.
hide	No borders are drawn around empty cells.

This property applies to the following formatting object:

table-cells

end-indent FOP_END_INDENT

`end-indent`

Default: 0pt

Specifies the distance between the right edge of the content area and right edge of the reference area.

This property creates an indention between a block area and the reference area containing it. The default value, “0pt”, specifies no indention. Negative numbers can be used to create an outdent.

The value for this property must be expressed in one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	region-body
list-block	simple-page-master
list-item	table

ends-row FOP_ENDS_ROW

`ends-row`

Default: false

Specifies whether a table cell ends a row.

This only applies to cells that are not contained in table-rows.

Possible values are described in the following table:

Value	Meaning
`false`	`This cell is not the end of the row.`
`true`	`This cell does end the row.`

This property applies to the following formatting object:

table-cell

See also: starts-row

external-destination FOP_END_EXTERNAL-DESTINATION

`external-destination`

Specifies an external resource for a link.

This property should be assigned a URI pointing to an external resource.

This property applies to the following formatting object:

basic-link

flow-name FOP_FLOW_NAME

`flow-name`

Defines the name of the flow.

This property must be initialized for each fo:flow and fo:static-content object. The name must be unique within a fo:page-sequence object. It is an error if the name is not unique or if it has not been initialized.

This property applies to the following formatting objects:

flow	static-content

font-family FOP_FONT_FAMILY

`font-family`

Specifies a prioritized list of font family names, generic family names, or both.

A font family is a group of fonts designed to be used in combination and exhibiting similarities in design. One member of the family may be italic, another bold, another condensed or using small caps. Font family names include “Helvetica”, “New Century Schoolbook”, and “Kyokasho ICA L”.

Generic font families are a fallback mechanism, a means of preserving some of the stylesheet author’s intent in the worst case when none of the specified fonts can be selected. For optimum typographic control, particular named fonts should be used in style sheets. The five generic font families are serif, sans-serif, cursive, fantasy, and monospace.

To specify more than one font-family, separate each family name by a comma. The higher priority items should appear to the left of the lower priority items. The following is an example:

```
<fo:block font-family="Courier, monospace">
  <xsl:apply-templates/>
</fo:block>
```

This property applies to the following formatting objects:

block	page-number
inline	title
leader	

See also: font-stretch, font-style, font-variant

`font-size`

Default: 12pt

Specifies the size of the font when set solid.

Negative values are not allowed. Although font size is normally expressed using points, such as “12pt”, the size may be assigned using any of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- block
- inline
- leader
- page-number
- title

See also: font-size-adjust

font-size-adjust FOP_FONT_SIZE_ADJUST

`font-size-adjust`

Specifies the ideal ratio between the x-height of a font and the size of the font.

Since not all fonts look the scale equivalently, this property allows authors to specify an aspect value for an element that will preserve the x-height of the first choice font in the substitute font.

Assign this value the aspect value of the first choice font to preserve the aspect ratio in a substitute font. By default, this value is 0 which does not preserve the aspect ratio.

This property only applies to languages that distinguish between uppercase and lowercase characters.

This property applies to the following formatting objects:

block	page-number
inline	title
leader	

See also: font-size

font-stretch FOP_FONT_STRETCH

`font-stretch`

Selects a normal, condensed, or extended face from a font family.

From narrowest to widest, the possible values are:

- ultra-condensed
- extra-condensed
- condensed
- semi-condensed
- normal
- semi-expanded
- expanded
- extra-expanded
- ultra-expanded

Two additional values allow changing the font face relative to its inherited value: wider and narrower. wider adjusts the value one position lower in the previous list, but not past ultra-expanded. narrower move adjusts the value one position higher in the list, but not past ultra-condensed. The default value is normal.

This property applies to the following formatting objects:

block	page-number
inline	title
leader	

See also: font-family

`font-style`

Default: normal

Selects a normal, italicized, or slanted face from a font family.

The following table describes the possible values for this property:

Value	Meaning
normal	Specifies a font labeled “normal” from the font family.
oblique	Specifies a font labeled “oblique”, “italicized”, or “slanted” from the font family.
italic	Specifies a font labeled “italic” from the font family. If an italic face isn’t available a font labeled “oblique” is selected.
backslant	Specifies a font labeled “backslant” from the font family.

This property applies to the following formatting objects:

- block
- inline
- leader
- page-number
- title

See also: font-family

font-variant FOP_FONT_VARIANT

`font-variant`

Default: normal

Specifies if a small-caps version of the current font should be used.

In a small-cap font, lowercase characters look similar to uppercase characters but in small size and slightly different proportions.

The possible values for this property are described in the following table:

Value	Meaning
normal	The small-caps version is not used. This is the default value.
small-caps	The small-caps version of the font is used if available.

This property applies to the following formatting objects:

block	page-number
inline	title
leader	

See also: font-family

font-weight FOP_FONT_WEIGHT

`font-weight`

Default: normal

Specifies the weight of the current font.

The possible values for this property are an integer from 100 to 900, from lighter to darker, or one of the following values:

Value	Meaning
normal	Same as a value of 400.
bold	Same as a value of 700.
bolder	Specifies the weight should be darker than the inherited weight, but not higher than 900.
lighter	Specifies the weight should be lighter than the inherited weight, but not lower than 100.

This property applies to the following formatting objects:

block	page-number
inline	title
leader	

height FOP_HEIGHT

`height`

Default: auto

Specifies the height of boxes generated by block-level areas.

The possible values for this property are described in the following table:

Value	Meaning
auto	The formatter calculates the height.
<length>	Specifies a fixed height.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- external-graphic
- table
- table-cell
- table-row

See also: width

hyphenate FOP_HYPHENATE

`hyphenate`

Default: false

Specifies whether hyphenation is allowed during line-breaking when formatting this object.

The possible values for this property are described in the following table:

Value	Meaning
false	Hyphenation is not allowed.
true	Hyphenation is allowed.

This property applies to the following formatting object:

- block

See also: hyphenation-character

hyphenation-character FOP_HYPHENATE_CHAR

`hyphenation-character`

Default: '-' (U+2010)

Specifies the Unicode character used when a hyphenation break occurs.

This property applies to the following formatting object:

block

See also: hyphenation

hyphenation-keep FOP_HYPHENATE_KEEP

`hyphenation-keep`

Default: auto

Controls whether hyphenation can be performed on the last line of a reference-area.

The possible values for this property are described in the following table:

Value	Meaning
auto	Hyphenation can be performed in any region.
column	Both parts of a hyphenated word must appear in the same column.
page	Both parts of a hyphenated word must appear on the same page.

This property applies to the following formatting object:

block

See also: hyphenation

hyphenation-push-character-count FOP_HYPHENATE_PUSH_CHARACTER_COUNT

`hyphenation-push-character-count`

Default: 2

Specifies the minimum number of characters in a hyphenated word after the hyphenation character.

The value of this property is the minimum number of characters pushed to the next line after the line ending with the hyphenation character. If ‘0’ or a negative integer, it is converted to the nearest integer greater than ‘0.’

This property applies to the following formatting object:

block

See also: hyphenation

hyphenation-remain-character-count
FOP_HYPHENATE_REMAIN_CHARACTER_COUNT

`hyphenation-remain-character-count`

Default: 2

Specifies the minimum number of characters in a hyphenated word before the hyphenation character.

This is the minimum number of characters in the word left on the line ending with the hyphenation character. The value can be any unsigned integer. If the value is not an integer greater than 0, it is converted to the nearest integer greater than 0.

This property applies to the following formatting object:

block

id FOP_ID

`id`

Specifies a unique identifier for this formatting object.

This property allows references to this object by other formatting objects, therefore the value of this property must be unique among all objects in the fo: namespace in the result tree. This property does not have a default value and must be initialized before being referenced.

This property applies to the following formatting objects:

basic-link	list-item-label
block	page-number
external-graphic	page-sequence
inline	table
leader	table-body
list-block	table-cell
list-item	table-row
list-item-body	

keep-together FOP_KEEP_TOGETHER

`keep-together`

Default: .within-line=auto, .within-column=auto, .within-page=auto

Imposes keep-together conditions on formatting objects.

The value of the keep-together property is a keep data type that consists of three components: within-line, within-column, and within-page. within-line applies to line-areas. within-column applies to column-areas. within-page applies to page-areas.

The possible values for each of the three components is listed in the following table:

Value	Meaning
auto	Does not impose any keep-together conditions.
always	Always attempts to keep the context together.

This property applies to the following formatting objects:

basic-link	list-item-body
block	list-item-label
inline	table
list-block	table-row
list-item	

See also: keep-with-next, keep-with-previous

`keep-with-next`

Default: .within-line=auto, .within-column=auto, .within-page=auto

Imposes keep-with-next conditions on formatting objects.

The value of the keep-with-next property is a keep data type that consists of three components: within-line, within-column, and within-page. within-line applies to line-areas. within-column applies to column-areas. within-page applies to page-areas.

The possible values for each of the three components is listed in the following table:

Value	Meaning
auto	Does not impose any keep-together conditions.
always	Always attempts to keep the context together.

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- list-block
- list-item
- page-number
- table
- table-row

See also: keep-together, keep-with-previous

keep-with-previous FOP_KEEP_WITH_PREVIOUS

`keep-with-previous`

Default: .within-line=auto, .within-column=auto, .within-page=auto

Imposes keep-with-previous conditions on formatting objects.

The value of the keep-with-previous property is a keep data type that consists of three components: within-line, within-column, and within-page. within-line applies to line-areas. within-column applies to column-areas. within-page applies to page-areas.

The possible values for each of the three components is listed in the following table:

Value	Meaning
auto	Does not impose any keep-together conditions.
always	Always attempts to keep the context together.

This property applies to the following formatting objects:

basic-link	list-item
block	page-number
external-graphic	table
inline	table-row
list-block	

See also: keep-together, keep-with-next

language FOP_LANGUAGE

`language`

Default: none

Specifies the language to be used with locale specific services.

Locale specific services include line-justification, line-breaking, and hyphenation. The possible values for this property are described in the following table:

Value	Meaning
none	Indicates that the language is not significant in formatting this document.
<language>	A language specifier in conformance with RFC1766. RFC1766 can be found at http://www.ietf.org/rfc/rfc1766.txt.

This property applies to the following formatting object:

block

leader-length FOP_LEADER_LENGTH

`leader-length`

Default: .minimum=0pt, .optimum=12.0pt, .maximum=100%

Specifies the minimum, optimum, and maximum length of an fo:leader.

The value of the leader-length property is a length-range data type. The length-range data type has three components: minimum, optimum, and maximum. minimum sets the minimum length for a leader. optimum sets the leader's optimum length. maximum sets the maximum length of the leader.

The value assigned to each component must be in one of the following terms:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

leader

See also: leader-pattern

leader-pattern FOP_LEADER_PATTERN

`leader-pattern`

Default: space

Specifies how to fill in the leader.

The possible values for this property are in the following table:

Value	Meaning
space	Fill leader with blank space.
rule	Fill leader with a rule (line).
Dots	Fill leader with a repeating sequence of dots.
use-content	Fill leader with a repeating pattern as specified by the children of the fo:leader.

This property applies to the following formatting object:

leader

See also: leader-length, rule-thickness

`letter-spacing`

Default: normal

Specifies spacing behavior between text characters.

This possible values for this property are described in the following table:

Value	Meaning
normal	Specifies to use the normal spacing for the current font.
<length>	Indicates inter-character space in addition to the default space between characters.

If an inter-character space is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

leader page-number

line-height FOP_LINE_HEIGHT

`line-height`

Default: normal

Specifies the line height for generated boxes.

If the property is set on a block-level element whose content is composed of inline-level elements, it specifies the minimal height of each generated inline box. If the property is set on an inline-level element, it specifies the exact height of each box generated by the element. When an element contains text that is rendered in more than one font, the line-height is determined according to the largest font size.

The possible values for this property are described in the following table:

Value	Meaning
normal	Line height is base on the current font.
<length>	The box height is set to this length. Negative values are illegal.

If an explicit length is used, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- page-number
- title

See also: line-height

line-height-shift-adjustment FOP_LINE_HEIGHT_SHIFT_ADJUSTMENT

`line-height-shift-adjustment`

Default: consider-shifts

Controls whether the line-height is adjusted for content that has a baseline-shift.

A baseline is an imaginary line running across the bottom and top of each line. Subscript and superscript letters cause these baselines to shift. This property specifies if the line height should accommodate subscript and superscript characters.

The possible values for this property are contained in the following table:

Value	Meaning
consider-shifts	Include the adjusted top-edge and bottom-edge of any characters that have a baseline-shift when determining line height.
disregard-shifts	Include the unshifted top-edge and bottom-edge of any characters that have a baseline-shift when determining line height.

This property applies to the following formatting object:

- basic-link
- block
- external-graphic
- inline
- leader
- page-number
- title

See also: line-height

margin-bottom FOP_MARGIN_BOTTOM

margin-bottom

Default: 0pt

Sets the bottom margin width of an area.

The margin must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- block
- list-block
- list-item
- region-body
- simple-page-master
- table

See also: margin-left, margin-right, margin-top

`margin-left`

Default: 0pt

Sets the left margin width of an area.

The margin must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	region-body
list-block	simple-page-master
list-item	table

See also: margin-bottom, margin-right, margin-top

margin-right FOP_MARGIN_RIGHT

`margin-right`

Default: 0pt

Sets the right margin width of an area.

The margin must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	region-body
list-block	simple-page-master
list-item	table

See also: margin-bottom, margin-left, margin-top

margin-top FOP_MARGIN_TOP

`margin-top`

Default: 0pt

Sets the top margin width of an area.

The margin must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	region-body
list-block	simple-page-master
list-item	table

See also: margin-bottom, margin-left, margin-right

master-name FOP_MASTER_NAME

`master-name`

Identifies or selects a master.

This property is used to uniquely identify a fo:simple-page-master or fo:page-sequence-master and allow other objects to reference those objects. The name must be unique across all page-masters and page-sequence-masters.

This property applies to the following formatting objects:

- page-sequence
- page-sequence-master
- repeatable-page-master-reference
- simple-page-master
- single-page-master-reference

number-columns-repeated FOP_NUMBER_COLUMNS_REPEATED

`number-columns-repeated`

Default: 1

Specifies the table-column specification should be repeated.

This property has the same effect as repeating the fo:table-column formatting object in the result tree. The value must be positive integer. If the value is negative or non-integer, it will be rounded to the next integer greater than 0.

This property applies to the following formatting object:

- table-column

See also: column-number

odd-or-even FOP_ODD_OR_EVEN

`odd-or-even`

Default: any

Defines a condition for page-master selection based on an even or odd page number.

This property determines if the referenced page-master is eligible for selection at this point in the page-sequence.

The following table contains the possible values for this property:

Value	Meaning
odd	This master is eligible for selection if the page number is odd.
even	This master is eligible for selection if the page number is even.
any	This master is eligible for selection regardless of whether the page number is odd or even.

Note that the page numbers in the previous table refers to the page number of the generated pages.

This property applies to the following formatting object:

conditional-page-master-reference

See also: blank-or-not-blank, page-position

orphans FOP_ORPHANS

`orphans`

Default: 2

Specifies the minimum number of lines of a paragraph that must be left at the bottom of a page.

This property applies to the following formatting object:

block

See also: widows

overflow FOP_OVERFLOW

`overflow`

Default: auto

Specifies whether the content of a block-level element is clipped when it overflows the element's box.

The possible values for this property as shown in the following table:

Value	Meaning
auto	The formatter decides what gets clipped.
hidden	Indicates that the content is clipped and that no scrolling mechanism should be provided to view the content outside the clipping region.
scroll	Indicates the content is clipped and that if the user agent uses a scrolling mechanism that is visible on the screen (such as a scroll bar), that mechanism should be displayed for a box whether or not any of its content is clipped. This avoids any problem with scrollbars appearing and disappearing in a dynamic environment. When this value is specified and the target medium is not interactive overflowing content should be printed.
visible	Indicates that content is not clipped and may be rendered outside the box.

This property applies to the following formatting objects:

external-graphic	region-body
region-after	region-end
region-before	region-start

padding-after FOP_PADDING_AFTER

`padding-after`

Default: 0pt

Specifies the width of the padding on the after-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

padding-before FOP_PADDING_BEFORE

`padding-before`

Default: 0pt

Specifies the width of the padding on the before-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

basic-link	region-before
block	region-body
external-graphic	region-end
inline	region-start
leader	table
list-block	table-body
list-item	table-cell
page-number	title
region-after	

`padding-bottom`

Default: 0pt

Specifies the width of the padding on the bottom-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`padding-end`

Default: 0pt

Specifies the width of the padding on the end-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`padding-left`

Default: 0pt

Specifies the width of the padding on the left-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`padding-right`

Default: 0pt

Specifies the width of the padding on the right-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

`padding-start`

Default: 0pt

Specifies the width of the padding on the start-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

padding-top FOP_PADDING_TOP

`padding-top`

Default: 0pt

Specifies the width of the padding on the top-edge of a block-area or inline-area.

The padding width must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

- basic-link
- block
- external-graphic
- inline
- leader
- list-block
- list-item
- page-number
- region-after
- region-before
- region-body
- region-end
- region-start
- table
- table-body
- table-cell
- title

page-height FOP_PAGE_HEIGHT

`page-height`

Default: auto

Specifies the height of a page.

The possible values for this property are described in the following table:

Value	Meaning
auto	Indicates that the formatter will determine the page height.
<length>	Specifies a fixed height for the page.

If a fixed height is specified, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

simple-page-master

See also: page-width

page-position FOP_PAGE_POSITION

`page-position`

Default: any

Defines a condition for page-master selection based on page position.

The following table contains the possible values for this property:

Value	Meaning
first	This master is eligible for selection if this is the first page in the page-sequence.
last	This master is eligible for selection if this is the last page in the page-sequence.
rest	This master is eligible for selection if this is not the first page nor the last page in the page-sequence.
any	This master is eligible for selection regardless of page positioning within the page-sequence.

Note that the page numbers in the previous page refers to the generated page numbers.

This property applies to the following formatting objects:

conditional-page-master-reference

See also: blank-or-not-blank, odd-or-even

`page-width`

Default: auto

Specifies the width of a page.

The possible values for this property are described in the following table:

Value	Meaning
auto	Indicates that the formatter will determine the page height.
<length>	Specifies a fixed width for the page.

If a fixed width is specified, it must use one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

simple-page-master

See also: page-height

`provisional-distance-between-starts`

Default: 24.0pt

Specifies the provisional distance between the start-indent of the list-item-label and the start-indent of the list-item-body.

The distance must be specified using one of the following units::

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

list-block

See also: provisional-label-separation

provisional-label-separation FOP_PROVISIONAL_LABEL_SEPARATION

`provisional-label-separation`

Default: 6.0pt

Specifies the provisional distance between the end of the list-item-label and the start of the list-item-body.

The distance must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

list-block

See also: provisional-distance-between-starts

region-name FOP_REGION_NAME

`region-name`

Identifies a region within a simple-page-master.

The value of region-name must be a unique name or one of the following predefined values:

Value	Description
xsl-region-body	Reserved region-name for use as default name of fo:region-body. This name may not be used on any other class of region.
xsl-region-start	Reserved region-name for use as default name of fo:region-start. This name may not be used on any other class of region.
xsl-region-end	Reserved region-name for use as default name of fo:region-end. This name may not be used on any other class of region.
xsl-region-before	Reserved region-name for use as default name of fo:region-before. This name may not be used on any other class of region.
xsl-region-after	Reserved region-name for use as default name of fo:region-after. This name may not be used on any other class of region.
xsl-before-float-separator	Reserved name for content to be used as a separator at the end-edge of a before-float-reference-area. This name may not be used on any other class of region.
xsl-footnote-separator	Reserved name for content to be used as a separator at the start-edge of a footnote-reference-area. This name may not be used on any other class of region.

This property applies to the following formatting objects:

region-after
region-before
region-body
region-end
region-start

`rule-thickness`

Default: 1.0pt

Specifies the overall thickness of the rule.

This property applies only if the leader-pattern property is specified as “rule.”

The thickness must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

leader

See also: leader, leader-pattern

scaling FOP_SCALING

`scaling`

Default: uniform

Specifies whether scaling needs to preserve the intrinsic aspect ratio.

Possible values for this property are shown in the following table:

Value	Meaning
uniform	Scaling should preserve the aspect ratio.
non-uniform	Scaling need not preserve the aspect ratio.

This property applies to the following formatting object:

external-graphic

`scaling-method`

Default: auto

Indicates a preference of scaling or sizing when formatting bitmapped graphics.

The allowable values for this property are listed in the following table:

Value	Meaning
auto	The formatter chooses the scaling method.
integer-pixels	The image is scaled such that each pixel in the original image is scaled to the nearest integer number of device-pixels that yields an image less-then-or-equal-to the image size derived from the content-height, content-width, and scaling properties.
resample-any-method	The image is resampled to provide an image that fills the size derived from the content-height, content-width, and scaling properties.

This property applies to the following formatting object:

external-graphic

show-destination FOP_SHOW_DESTINATION

`show-destination`

Default: replace

Specifies where the destination resource should be displayed.

The following table explains the possible values for this property:

Value	Meaning
replace	The current document view is replaced. However, if the destination area is already available in a page or region, those areas should simply be moved or scrolled into sight.
new	A new document view is always opened.

This property applies to the following formatting object:

basic-link

`space-after`

Default: 0pt

Specifies the amount of space after the block-level areas generated by this formatting object.

The spacing distance must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	list-item	simple-page-master
list-block	region-body	table

See also: space-before

space-before FOP_SPACE_BEFORE

`space-before`

Default: 0pt

Specifies the amount of space before the block-level areas generated by this formatting object.

The spacing distance must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block	list-item	simple-page-master
list-block	region-body	table

See also: space-after

space-end FOP_SPACE_END

`space-end`

Default: 0pt

Specifies the amount of space after the inline areas generated by this formatting object.

This property applies to the following formatting objects:

basic-link	inline	page-number
external-graphic	leader	title

See also: space-start

space-start FOP_SPACE_START

`space-start`

Default: 0pt

Specifies the amount of space before the inline areas generated by this formatting object.

The spacing distance must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

basic-link	inline	page-number
external-graphic	leader	title

See also: space-end

`src`

Specifies the URI reference to an external resource.

This property applies to the following formatting object:

external-graphic

start-indent FOP_START_INDENT

`start-indent`

Default: 0pt

Specifies the distance between the left edge of the content area and left edge of the reference area.

This property creates an indention between a block area and the reference area containing it. The default value, “0pt”, specifies no indention. Negative numbers can be used to create an outdent.

The value for this property must be expressed in one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

block region-body simple-page-master

See also: end-indent

starts-row FOP_STARTS_ROW

`starts-row`

Specifies whether this cell starts a row.

This property only applies to cells that are not contained in a table-row.

Possible values are described in the following table:

Value	Meaning
false	This cell is not the start of the row.
true	This cell does start the row.

This property applies to the following formatting object:

table-cell

See also: ends-row

text-align FOP_TEXT_ALIGN

`text-align`

Default: start

Describes how inline content of a block is aligned.

The possible values for this property are shown in the following table:

Value	Meaning
center	The content is to be centered in the inline-progression-direction.
end	The content is to be aligned on the end-edge in the inline-progression-direction.
justify	The contents is to be expanded to fill the available width in the inline-progression-direction.
inside	If the page binding edge is on the start-edge, the alignment will be start. If the binding is the end-edge, the alignment will be end. If neither, use start alignment.
outside	If the page binding edge is on the start-edge, the alignment will be end. If the binding is the end-edge, the alignment will be start. If neither, use end alignment.
start	Specifies that the content is to be aligned on the start-edge in the inline-progression-direction.

This property applies to the following formatting objects:

block

external-graphic

See also: text-align-last

`text-align-last`

Default: relative

Specifies the alignment of the last line-area in a block, and any lines in the block ending in U+000A.

The possible values are shown in the following table:

Value	Meaning
center	Specifies that the contents is to be centered in the inline-progression-direction.
end	Specifies that the content is to be aligned on the end-edge in the inline-progression-direction.
justify	Specifies that the contents is to be expanded to fill the available width in the inline-progression-direction.
inside	If the page binding edge is on the start-edge, the alignment will be start. If the binding is the end-edge, the alignment will be end. If neither, use start-side.
outside	If the page binding edge is on the start-edge, the alignment will be end. If the binding is the end-edge the alignment will be start. If neither, use end alignment.
relative	If text-align is justify, then the alignment of the last line, and of any line ending in U+000A, will be start. If text-align is not justify, text-align-last will use the value of text-align.
start	Specifies that the content is to be aligned on the start-edge in the inline-progression-direction.

This property applies to the following formatting object:

block

See also: text-align

`text-indent`

Default: 0pt

Specifies the indentation of the first line of text in a block.

The indentation distance must be expressed in one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting object:

block

widows FOP_WIDOWS

`widows`

Default: 2

Specifies the minimum number of lines of a paragraph that must be left at the bottom of a page.

This property applies to the following formatting object:

block

See also: orphans

`width`

Default: auto

Specifies the content width of boxes generated by block-level and replaced elements.

The possible values for this property are described in the following table:

Value	Meaning
auto	Indicates that the formatter will determine the width.
Length	Specifies a fixed width.

If a fixed width is specified, it must be expressed using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

external-graphic table table-cell

See also: height

word-spacing FOP_WORD_SPACING

`word-spacing`

Default: normal

Specifies spacing behavior between words.

The possible values for this property are described in the following table:

Value	Meaning
normal	Indicates that the formatter will determine the word spacing based on the current font.
<length>	Specifies an inter-word space in addition to the default space between words.

If an inter-word space is used, it must be specified using one of the following units:

Unit	Description
in	Inches; 1 inch is equal to 2.54 centimeters
cm	Centimeters
mm	Millimeters
pt	Points; equal to 1/72th of an inch
pc	Picas; 1 pica is equal to 12 points

This property applies to the following formatting objects:

leader page-number

wrap-option FOP_WRAP_OPTION

`wrap-option`

Default: wrap

Specifies how line-wrapping of the formatting object's content is to be handled.

The possible values for this property are shown in the following table:

Value	Meaning
nowrap	No line-wrapping will be performed. In the case when lines are longer than the available width of the content-rectangle, the overflow will be treated in accordance with the "overflow" property specified on the reference-area.
wrap	Line-breaking will occur if the line overflows the available block width. No special markers or other treatment will occur.

This property applies to the following formatting object:

block

See also: overflow

W3C Copyright info

http://www.w3.org/TR/xsl/

Copyright© 2000 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

This document is a W3C Candidate Recommendation

Chapter 10: Appendix

This chapter provides information on a set of miscellaneous topics:

- Error codes returned during DOM validation.
- Error codes returned with instances of TXpDOMException.
- XPath functions supported by the XMLPartner DOM.
- Predefined colors for formatting properties.
- XSLT and XSL-FO tag codes.
- XSL-FO property tag codes.
- Glossary of terms.

DOM Validation Error Codes

The ValidateDocument method of the TXpObjModel class validates a document's content against its Document Type Definition (DTD). If an error is found during the validation process, the ValidateDocument method raises the OnInvalidDocument event of the TXpObjModel class. The wCode parameter of the OnInvalidDocument event identifies the type of error encountered. Table 10.1 lists the error codes that may be assigned to the wCode parameter of this event.

Table 10.1: *DOM validation error codes*

Constant	Value	Meaning
V_NODTD	1	The document does not contain a DTD or a reference to an external DTD. A document must have a DTD in order for the document to be validated.
V_BADDTD	2	The DTD has a bad structure. This could occur when an element is identified as having mixed content but no definition was found for the possible elements within its content.
V_NODOCUMENT	3	The object model does not contain a document.
V_MISMATCH	4	The document name does not match the DTD name.
V_ELEMENTNOTDEFINED	5	An element name found in the document is not defined in the DTD.
V_ELEMENTNOTEMPTY	6	An element was declared as empty, but it is not empty.
V_ELEMENTNOTCONTENT	7	An element contains a child element, but the parent element's definition does not allow that type of child element.
V_ELEMENTCHILDMISSING	8	A required child element is missing from an element.
V_ELEMENTCHILDNOTEXPECTED	9	A child element was encountered, but not expected.
V_ELEMENTATTRNOTDEFINED	10	An element attribute was found, but it is not defined in the DTD.

Table 10.1: *DOM validation error codes (continued)*

V_ATTRIBUTEFIXED	11	An attribute is declared as having a fixed value, but the document assigns some other value to the element.
V_ATTRIBUTENOTENUM	12	An enumerated attribute is assigned a value that is not one of the specified values in the DTD.
V_ATTRIBUTEREQUIRED	13	An attribute is marked as required in the DTD, but an instance of the element within the document does not have a value for the attribute.

Exception Error Codes

If an instance of TXpDOMException is raised during the evaluation of a document, the error code associated with the instance is available through its Code property. Table 10.2 lists the possible values for the Code property and the meaning of each value.

Table 10.2: *Exception error codes*

Value	Meaning
ecNone	No exception has occurred. This is the default value for the Code property and should not be encountered during normal use.
ecIndexSizeErr	Raised if the specified offset is negative or greater than the number of characters available in the operation, or if a count is negative.
ecWStringSizeErr	Raised if the specified range of text does not fit in a DOMString variable.
ecHierarchyRequestErr	Raised if a node is inserted somewhere it doesn't belong.
ecWrongDocumentErr	Raised if a node involved in an operation does not have that same owner document as the other node. For example, this error is raised when appending a node owned by document A into the child list of an element owned by document B.
ecInvalidNameErr	Raised if an invalid character is specified, such as in a name.
ecNoDataAllowedErr	Raised if data is specified for a node which does not support data.
ecNoModificationAllowedErr	Raised if an attempt is made to modify a read-only object.
ecNotFoundErr	Raised if an attempt was made to reference a node in a context in which it does not exist. For example, using the InsertBefore method of the TXpNode class to insert a node before a node that is not a child of the current element will cause this error to be raised.

Table 10.2: *Exception error codes (continued)*

`ecNotSupportedErr`	Raised if the implementation does not support the type of object requested.
`ecInuseAttributeErr`	Raised if an attribute is already used by another element. Clone the attribute before it is re-used in another element.
`ecInvalidParamErr`	Raised if an invalid parameter is specified for an operation.

XPath Functions

The XMLPartner Document Object Model (DOM) supports the use of XPath 1.0 for the searching of an XML document. In addition, XMLPartner also supports some of the functions declared in the XSLT specification. Table 10.3 lists the functions supported by the XMLPartner DOM. For full information about each function, see the XPath 1.0 recommendation at http://www.w3.org/TR/xpath.

Table 10.3: *XPath functions*

Function	Purpose
boolean	Converts its argument to a Boolean.
ceiling	Returns the smallest integer that is greater than or equal to the numeric value of the argument.
concat	Returns the concatenation of its arguments.
contains	Returns True if the first argument string contains the second argument string.
count	Returns the number of nodes in a node set.
false	Returns false.
floor	Returns the largest integer that is less than or equal to the numeric value of the argument.
function-available	Returns True if a specific function is available. Use this to test for standard or extension functions.
generate-id	Generates a unique ID for a node. The unique ID is returned as a string value. The ID returned by XMLPartner is the node's level code.
id	Searches for an element having a specific value for its ID attribute.
lang	Returns True if the language of the context node matches the language specified in the argument.
last	Returns the value of the context size. For example, if the current context is a list of nodes then this function returns the number of nodes.
local-name	If a node belongs to a namespace then this function returns the local portion of the node's name (i.e., the portion after the semi-colon). Otherwise, it returns the node's full name.
name	Returns the qualified name of a node.

Table 10.3: *XPath functions (continued)*

namespace-uri	Returns the namespace URI of a node.
normalize-space	Returns a copy of the string with normalized such that leading and trailing white space has been removed and sequences of white space characters have been replaced by a single space.
not	Returns True if its argument is False or False if its argument is True.
number	Converts its argument to a number.
position	Returns the value of the current position within the current context. For example, if the current context is a list of nodes then this function returns the number assigned to the current node in the list.
round	Returns the integer that is closest to the numeric value of the argument. XMLPartner implements this function using the Round function found in CLX or VCL.
starts-with	Returns True if the first argument string starts with the second argument string.
string	Converts a number, boolean value, or other object to a string.
string-length	Returns the number of characters in a string.
substring	Returns a substring of a string.
substring-after	Returns the substring of the first argument string following the first occurrence of the second argument string in the first argument string. For example, “substring-after("2001/06/18","/")” returns “06/18”.
substring-before	Returns the substring of the first argument string that precedes the first occurrence of the second argument string in the first argument string. For example, “substring-before("2001/06/18","/")” returns “2001”.
sum	Returns the sum of each node in the node set where the string value of each node is converted to a number.
system-property	Returns the value of a system property.
translate	Replace occurrences of characters with another character.
true	Returns True.

Predefined colors for formatting properties

The color formatting properties (background-color, color, etc.) require a predefined name of a color or a hexadecimal RGB value. RGB values are formed by a '#' character followed by three two-digit hexadecimal values. The first two digits define the amount (00-FF) of the color red in the color value, digits three and four define the amount of green in the color value, and the last two digits define the amount of blue. As an example, the hexadecimal RGB value for the color yellow is "#FFFF00".

Any of the color names or hexadecimal RGB values shown in Table 10.4 can be used as a color values. Other color codes may also be used.

Table 10.4: *Predefined colors for formatting properties*

Name	Hexadecimal RGB Value
aliceblue	#F0F8FF
antiquewhite	#FAEBD7
auqa	#00FFFF
auqamarine	#7FFFD4
azure	#F0FFFF
beige	#F5F5DC
bisque	#FFE4C4
blanchdalmond	#FFEBCD
blue	#0000FF
blueviolet	#8A2BE2
brown	#A52A2A
burlywood	#DEB887
cadetblue	#5F9EA0
chartreuse	#7FFF00
chocolate	#D2691E
coral	#FF7F50
cornflowerblue	#6495ED
cornsilk	#FFF8DC
crimson	#DC143C
cyan	#00FFFF

Table 10.4: *Predefined colors for formatting properties (continued)*

Name	Hexadecimal RGB Value
darkblue	#00008B
darkcyan	#008B8B
darkgoldenrod	#B8860B
darkgray	#A9A9A9
darkgreen	#006400
darkkhaki	#BDB76B
darkmagenta	#8B008B
darkolivegreen	#556B2F
darkorange	#FF8C00
darkorchid	#9932CC
darkred	#8B0000
darksalmon	#E9967A
darkseagreen	#8FBC8F
darkslateblue	#483D8B
darkslategray	#2F4F4F
darkturquoise	#00CED1
darkviolet	#9400D3
deeppink	#FF1493
deepskyblue	#00BFFF
dimgray	#696969
dodgerblue	#1E90FF
firebrick	#B22222
floralwhite	#FFFAF0
forestgreen	#228B22
fuchsia	#FF00FF
gainsboro	#DCDCDC
ghostwhite	#F8F8FF
gold	#FFD700
goldenrod	#DAA520

Table 10.4: *Predefined colors for formatting properties (continued)*

Name	Hexadecimal RGB Value
gray	#808080
green	#008000
greenyellow	#ADFF2F
honeydew	#F0FFF0
hotpink	#FF69B4
indianred	#CD5C5C
indigo	#4B0082
ivory	#FFFFF0
khaki	#F0E68C
lavender	#E6E6FA
laverblush	#FFF0F5
lawngreen	#7CFC00
lemonchiffon	#FFFACD
lightblue	#ADD8E6
lightcoral	#F08080
lightcyan	#E0FFFF
lightgoldenrodyellow	#FAFAD2
lightgreen	#90EE90
lightgray	#D3D3D3
lightpink	#FFB6C1
lightsalmon	#FFA07A
lightseagreen	#20B2AA
lightskyblue	#87CEFA
lightslategray	#778899
lightsteelblue	#B0C4DE
lightyellow	#FFFFE0
lime	#00FF00
limegreen	#32CD32
linen	#FAF0E6

Table 10.4: *Predefined colors for formatting properties (continued)*

Name	Hexadecimal RGB Value
magenta	#FF00FF
maroon	#800000
mediumaquamarine	#66CDAA
mediumblue	#0000CD
mediumorchid	#BA55D3
mediumpurple	#9370DB
mediumseagreen	#3CB371
mediumslateblue	#7B68EE
mediumspringgreen	#00FA9A
mediumturquoise	#48D1CC
mediumvioletred	#C71585
midnightblue	#191970
mintcream	#F5FFFA
mistyrose	#FFE4E1
moccasin	#FFE4B5
navajowhite	#FFDEAD
navy	#000080
oldlace	#FDF5E6
olive	#808000
olivedrab	#6B8E23
orange	#FFA500
orangered	#FF4500
orchid	#DA70D6
palegoldenrod	#EEE8AA
palegreen	#98FB98
paleturquoise	#AFEEEE
palevioletred	#DB7093
papayawhip	#FFEFD5
peachpuff	#FFDAB9

Table 10.4: *Predefined colors for formatting properties (continued)*

Name	Hexadecimal RGB Value
peru	#CD853F
pink	#FFC0CB
plum	#DDA0DD
powderblue	#B0E0E6
purple	#800080
red	#FF0000
rosybrown	#BC8F8F
royalblue	#4169E1
saddlebrown	#8B4513
salmon	#FA8072
sandybrown	#F4A460
seagreen	#2E8B57
seashell	#FFF5EE
sienna	#A0522D
silver	#C0C0C0
skyblue	#87CEEB
slateblue	#6A5ACD
slategray	#708090
snow	#FFFAFA
springgreen	#00FF7F
steelblue	#4682B4
tan	#D2B48C
teal	#008080
thistle	#D8BFD8
tomato	#FF6347
turquoise	#40E0D0
violet	#EE82EE
wheat	#F5DEB3
white	#FFFFFF

Table 10.4: *Predefined colors for formatting properties (continued)*

Name	Hexadecimal RGB Value
whitesmoke	#F5F5F5
yellow	#FFFF00
yellowgreen	#9ACD32

XSLT and XSL-FO tag codes

Some of the XSL processor's events return or expect a tag code for a corresponding XSLT command or formatting object. Table 10.5 lists these tag codes and their value. See the next section for formatting property tag codes.

Table 10.5: *XSLT and XSL-FO tag codes*

Tag Code	Value
UNKNOWN	-1
TAG_APPLY_IMPORTS	1
TAG_APPLY_TEMPLATES	2
TAG_ATTRIBUTE	3
TAG_ATTRIBUTE_SET	4
TAG_CALL_TEMPLATE	5
TAG_CHOOSE	6
TAG_COMMENT	7
TAG_COPY	8
TAG_COPY_OF	9
TAG_DECIMAL_FORMAT	10
TAG_ELEMENT	11
TAG_FALLBACK	12
TAG_FOR_EACH	13
TAG_IF	14
TAG_IMPORT	15
TAG_INCLUDE	16
TAG_KEY	17
TAG_MESSAGE	18
TAG_NAMESPACE_ALIAS	19
TAG_NUMBER	20
TAG_OTHERWISE	21
TAG_OUTPUT	22
TAG_PARAM	23

Table 10.5: *XSLT and XSL-FO tag codes (continued)*

Tag Code	Value
TAG_PRESERVE_SPACE	24
TAG_PROCESSING_INSTRUCTION	25
TAG_SORT	26
TAG_STRIP_SPACE	27
TAG_STYLESHEET	28
TAG_TEMPLATE	29
TAG_TEXT	30
TAG_TRANSFORM	31
TAG_VALUE_OF	32
TAG_VARIABLE	33
TAG_WHEN	34
TAG_WITH_PARAM	35
FO_BASIC_LINK	100
FO_BLOCK	101
FO_CONDITIONAL_PAGE_MASTER_REFERENCE	102
FO_EXTERNAL_GRAPHIC	103
FO_FLOW	104
FO_INLINE	105
FO_LAYOUT_MASTER_SET	106
FO_LEADER	107
FO_LIST_BLOCK	108
FO_LIST_ITEM	109
FO_LIST_ITEM_BODY	110
FO_LIST_ITEM_LABEL	111
FO_PAGE_NUMBER	112
FO_PAGE_SEQUENCE	113
FO_PAGE_SEQUENCE_MASTER	114
FO_REGION_AFTER	115
FO_REGION_BEFORE	116

Table 10.5: *XSLT and XSL-FO tag codes (continued)*

Tag Code	Value
FO_REGION_BODY	117
FO_REGION_END	118
FO_REGION_START	119
FO_REPEATABLE_PAGE_MASTER_ALTERNATIVES	120
FO_REPEATABLE_PAGE_MASTER_REFERENCE	121
FO_ROOT	122
FO_SIMPLE_PAGE_MASTER	123
FO_SINGLE_PAGE_MASTER_REFERENCE	124
FO_STATIC_CONTENT	125
FO_TABLE	126
FO_TABLE_BODY	127
FO_TABLE_CELL	128
FO_TABLE_COLUMN	129
FO_TABLE_ROW	130
FO_TITLE	131

XSL-FO Property Tag Codes

The XSL processor's OnFoProperty event includes a tag code for the formatting property found in the XSL stylesheet. Table 10.6 shows the tags codes and their value.

Table 10.6: *XSL-FO property tag codes*

Tag Code	Value
FOP_ALIGNMENT_ADJUST	300
FOP_BACKGROUND_ATTACHMENT	301
FOP_BACKGROUND_COLOR	302
FOP_BACKGROUND_IMAGE	303
FOP_BACKGROUND_POSITION_H	304
FOP_BACKGROUND_POSITION_V	305
FOP_BLANK_OR_NOT_BLANK	306
FOP_BORDER_AFTER_COLOR	307
FOP_BORDER_AFTER_WIDTH	308
FOP_BORDER_BEFORE_COLOR	309
FOP_BORDER_BEFORE_WIDTH	310
FOP_BORDER_BOTTOM_COLOR	311
FOP_BORDER_BOTTOM_WIDTH	312
FOP_BORDER_END_COLOR	313
FOP_BORDER_END_WIDTH	314
FOP_BORDER_LEFT_COLOR	315
FOP_BORDER_LEFT_WIDTH	316
FOP_BORDER_RIGHT_COLOR	317
FOP_BORDER_RIGHT_WIDTH	318
FOP_BORDER_START_COLOR	319
FOP_BORDER_START_WIDTH	320
FOP_BORDER_TOP_COLOR	321
FOP_BORDER_TOP_WIDTH	322
FOP_BREAK_AFTER	323
FOP_BREAK_BEFORE	324

Table 10.6: *XSL-FO property tag codes (continued)*

Tag Code	Value
FOP_COLOR	325
FOP_COLUMN_NUMBER	326
FOP_COLUMN_WIDTH	327
FOP_CONTENT_HEIGHT	328
FOP_CONTENT_WIDTH	329
FOP_DISPLAY_ALIGN	330
FOP_EMPTY_CELLS	331
FOP_END_INDENT	332
FOP_ENDS_ROW	333
FOP_EXTERNAL_DESTINATION	334
FOP_FLOW_NAME	335
FOP_FONT_FAMILY	336
FOP_FONT_SIZE	337
FOP_FONT_SIZE_ADJUST	338
FOP_FONT_STRETCH	339
FOP_FONT_STYLE	340
FOP_FONT_VARIANT	341
FOP_FONT_WEIGHT	342
FOP_HEIGHT	343
FOP_HYPHENATE	344
FOP_HYPHENATE_CHAR	345
FOP_HYPHENATE_KEEP	346
FOP_HYPHENATE_PUSH_CHARACTER_COUNT	347
FOP_HYPHENATE_REMAIN_CHARACTER_COUNT	348
FOP_ID	349
FOP_KEEP_TOGETHER	350
FOP_KEEP_WITH_NEXT	351
FOP_KEEP_WITH_PREVIOUS	352
FOP_LANGUAGE	353

Table 10.6: *XSL-FO property tag codes (continued)*

Tag Code	Value
FOP_LEADER_LENGTH	354
FOP_LEADER_PATTERN	355
FOP_LETTER_SPACING	356
FOP_LINE_HEIGHT	357
FOP_LINE_HEIGHT_SHIFT_ADJUSTMENT	358
FOP_MARGIN_BOTTOM	359
FOP_MARGIN_LEFT	360
FOP_MARGIN_RIGHT	361
FOP_MARGIN_TOP	362
FOP_MASTER_NAME	363
FOP_NUMBER_COLUMNS_REPEATED	364
FOP_ODD_OR_EVEN	365
FOP_ORPHANS	366
FOP_OVERFLOW	367
FOP_PADDING_AFTER	368
FOP_PADDING_BEFORE	369
FOP_PADDING_BOTTOM	370
FOP_PADDING_END	371
FOP_PADDING_LEFT	372
FOP_PADDING_RIGHT	373
FOP_PADDING_START	374
FOP_PADDING_TOP	375
FOP_PAGE_HEIGHT	376
FOP_PAGE_POSITION	377
FOP_PAGE_WIDTH	378
FOP_PROVISIONAL_DISTANCE_BETWEEN_STARTS	379
FOP_PROVISIONAL_LABEL_SEPARATION	380
FOP_REGION_NAME	381
FOP_RULE_THICKNESS	382

Table 10.6: *XSL-FO property tag codes (continued)*

Tag Code	Value
FOP_SCALING	383
FOP_SCALING_METHOD	384
FOP_SHOW_DESTINATION	385
FOP_SPACE_AFTER	386
FOP_SPACE_BEFORE	387
FOP_SPACE_END	388
FOP_SPACE_START	389
FOP_SRC	390
FOP_START_INDENT	391
FOP_STARTS_ROW	392
FOP_TEXT_ALIGN	393
FOP_TEXT_ALIGN_LAST	394
FOP_TEXT_INDENT	395
FOP_WIDOWS	396
FOP_WIDTH	397
FOP_WORD_SPACING	398
FOP_WRAP_OPTION	399

Glossary

area

A node in an area tree. Areas represent rectangular portions of the transformed document that contain lines, text, paragraphs, and other portions of content.

area tree

Ordered tree created by a maintainer of formatting objects that represents a formatted document.

attribute

A qualifier for an element. An attribute has a name and a value. The value must be enclosed in quotation marks. For example, in the tag <book form="hardbound">, form is an attribute and “hardbound” is the value of the attribute.

attribute value template

An XPath expression nested within curly braces (i.e., ‘{ and ‘}’). The XSL processor evaluates the expression and uses its value as the name of the attribute. The resulting name must be a valid Qualified Name (i.e., QName) having a local name and an optional namespace prefix (e.g., “style”, “ams:distribClass”).

block areas

Areas in the tree created by formatting objects that are stacked from top to bottom on a page, like paragraphs.

character data

All text within the document that is not markup.

comment

Text within the document that may be used like source code comments. An XML processor ignores the contents of a comment.

content

The part of an XML document including and between the start and end tags of the document element (i.e., root element). In other words, content is everything after an XML document’s prolog.

CDATA section

A structural element whose content is ignored by the XML processor. CDATA sections are used to prevent processing of text or data that would otherwise be considered malformed XML. For example, it can be used to show examples of XML syntax or to contain HTML markup.

current node

The node with focus from the current nodelist while processing an XSL stylesheet. A template is always applied to the current node. Use xsl:for-each to retrieve a node list and iterate through each of its nodes.

current node list

The node list with focus while processing an XSL stylesheet. The current node list is retrieved using xsl:template or xsl:for-each.

current template rule

Concept of applying the current set of template rules to the current nodelist. The current template is usually determined with xsl:template, but can be overridden using xsl:apply-imports.

declaration

When present, this is the very first set of markup in an XML document that identifies it as XML. When present, the declaration must contain at least the following information: <?xml version="1.0"?>

document element

The top-level element of an XML document. Each XML document may have only one document element, often referred to as the root element. The document element contains the content of an XML document and follows the document's prolog.

Document Object Model (DOM)

As defined by the W3C, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of documents. The DOM provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. The XMLPartner implementation of DOM handles XML documents only.

Document Type Definition (DTD)

The markup that describes the grammar for one or more XML documents. Among other things, a DTD defines what elements and attributes may exist within a document and how those elements and attributes are structured. A DTD may be embedded within an XML document or an XML document may contain a reference to an external DTD.

DOM

See Document Object Model (DOM).

DTD

See Document Type Definition (DTD).

element

A logical structure within an XML document. A start tag and an end tag delimit an element. An element may contain other elements and content. For example, in the element "<dog>Scooter</dog>", "<dog>" is the start tag, "Scooter" is the content, and "</dog>" is the end tag.

entity

Markup describing a set of text that is to be used later in the document, much in the same way that a macro is used in the C language. An entity declaration contains a start tag, an entity name, some content, and an end tag. The entity declaration occurs in the DTD. The content of the XML document may contain entity references. When the document is processed, the XML processor may replace the entity references with the entity's content. An entity may be internal or external. The declaration of an internal entity contains the replacement text for entity references. The declaration of an external entity identifies an external physical storage object (i.e., a file) containing the entity's replacement text. Note that an entity's content may contain an entity reference to another entity.

entity reference

Markup that references an entity previously declared in the document's DTD. An XML processor may elect to replace entity references with the content of an entity. If it does so then the XML processor parses the entity's replacement text.

Extensible Markup Language (XML)

A subset of SGML that defines a standard for structuring data such that it can be easily exchanged and processed.

Extensible Style Language (XSL)

A language used to transform XML documents into formats, such as HTML, suitable for display in a Web browser.

flows

Contents of a formatted document. The contents are returned from the children of fo:flow and fo:static-content objects.

formatting

The process of preparing the result of an XSL transformation into a presentation form.

formatting object

Instructions that prepare the results of an XSL transformation for presentation. Formatting objects are defined as elements in an XSL stylesheet within the fo namespace.

formatting property

An attribute of a formatting object that provides the details of how a formatter prepares data for presentation.

HyperText Markup Language (HTML)

A markup language for describing the layout, formatting, and content of a Web page.

import precedence

The precedence assigned to xsl:stylesheet elements retrieved using xsl:import. The precedence determines which template rule should be applied to the current node list.

import tree

A structure containing the collection of stylesheets retrieved using xsl:import. Stylesheets are stored in the tree by precedence and in a way that ensures each stylesheet is processed after its imported children.

inline area

Areas in the tree created by formatting objects that are stacked from left to right and wrap downward on a page, like text.

literal result element

An element in an XSL stylesheet that does not belong to the XSLT namespace.

maintainer

An object responsible for maintaining an XSL result tree. In XMLPartner this is usually a filter component, but an application can be a maintainer by handling the XSL processor's events directly.

markup

Text within an XML document that defines the structure of the document. Markup consists of start tags, end tags, empty element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions.

metalanguage

A language used to describe the construction of another language. XML is a metalanguage because it defines the vocabulary (i.e., markup) that may be used in an XML document.

namespace

A method for qualifying element and attributes names used in XML documents. When the same element or attribute name is used among multiple XML documents but the meaning of the element or attribute name differs between the documents, a collision may occur. For example, the element "reader" in document A refers to a text scanner while an element of the same name in document B refers to a person participating in a literacy program. Namespaces allow element or attribute names to be associated with a URI thus imparting uniqueness and avoiding collisions.

namespace prefix

A shortname for a namespace URI. They associate an element or attribute name with a namespaceURI.

namespace URI

A URI that identifies an XML namespace.

page-viewport-area

Area tree node that represents a visible page in the transformed document. The page-viewport-area contains the region-viewport-areas.

prolog

The portion of the XML document preceding the content. The prolog may contain a declaration and a Document Type Definition (DTD).

region-viewport-area

Area tree node that represents one of the five visible regions of a page-viewport-area.

result tree

A structure containing the results of an XSLT transformation.

root element

See document element.

SGML

See Standard Generalized Markup Language (SGML).

Standard Generalized Markup Language (SGML)

A metalanguage used to define other markup languages. As a subset of SGML, the purpose of XML is to deliver SGML-like information over the Internet.

stylesheet

A document specifying the presentation for a class of XML documents.

tag

Sometimes referred to as markup, a tag is one or more characters used to define the structure of data within an XML document. For example, the element "<dog>Fido</dog>" contains the start tag "<dog>" and the end tag "</dog>".

top level element

An element that is direct child of a xsl:stylesheet element.

transformation

The process of converting the data in an XML document or a subset into a new form. In technical terms, the source tree (XML document) is transformed to create the result tree.

Unicode

A standard for the interchange, processing, and display of the written texts of the diverse languages of the world. Unicode represents each character with 16 bits which means that it can include all the world's languages, with the exception of ideographic languages.

Universal Resource Identifier (URI)

The way to address a resource on the Internet. URLs and URNs are types of URI. A URI indicates the mechanism to reach the resource (e.g., FTP, HTTP), the computer housing the resource, and the specific name of the resource on that computer.

Universal Resource Locator (URL)

The address of a file accessible on the Internet. A URL is a type of URI.

Universal Resource Name (URN)

A type of URI where the address of a resource where the name has persistent significance. The exact location (i.e., URL) of the resource may change but the resource may always be found by virtue of its URN.

validating processor

A processor that verifies the content of an XML document matches the vocabulary specified in the document's DTD.

viewport

A node of the formatted document's area tree that represents a rectangular section of the transformed document.

W3C

As described at http://www.w3c.org, an international organization created in October 1994 to lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability.

well-formed

XML that meets the requirements listed in the W3C XML 1.0 Recommendation. A well-formed XML document contains one or more elements and there is exactly one document element. Each of the parsed entities referenced within the document must also be well-formed.

XML

See Extensible Markup Language (XML).

XML Path Language (XPath)

A language for referencing information or assigning an address to information within an XML document. XPath uses expressions to navigate through an XML document, address sections of the document, and select sets of elements within the document.

XML processor

A software application that reads an XML document. All processors must verify an XML document is well-formed. Some processors also validate the content of the document against the document's DTD.

XSL

See Extensible Style Language (XSL).

XSL Transformation Language (XSLT)

A language used to transform XML documents into other XML documents.

Identifier Index

A

ActualCDATA 129
Add 140, 146
AddRef 110
alignment-adjust 392
AppendChild 110
AppendData 151
ApplyStyle 250
Attributes 110
AttType 201

B

background-attachment 393
background-color 394
background-image 394
background-position-horizontal 395
background-position-vertical 396
basic-link 367
blank-or-not-blank 398
block 368
BodyOnly 283
border-after-color 398
border-after-width 399
border-before-color 400
border-before-width 400
border-bottom-color 401
border-bottom-width 402
border-end-color 403
border-end-width 404
border-left-color 405
border-left-width 405
border-right-color 406
border-right-width 407
border-start-color 408
border-start-width 409
border-top-color 410
border-top-width 410
break-after 411
break-before 412
BufferSize 53, 86

C

ceXxx 55
ChildNodes 111
ClearDocument 86
ClearVariables 250
CloneNode 111
Code 102
color 413
column-number 414
column-width 414
conditional-page-master-reference 369
CondType 211
content-height 415
ContentType 188
content-width 415
CopyList 140
CreateAttribute 129
CreateAttributeNS 129
CreateCDATASection 130
CreateChildCDataSection 158
CreateChildElement 158
CreateChildText 158
CreateComment 130
CreateDocument 105
CreateDocumentFragment 130
CreateDocumentType 105, 131
CreateDTDAttDefinition 197
CreateDTDAttList 184
CreateDTDConditional 184
CreateDTDElement 185
CreateDTDElementContent 190
CreateDTDElementGroup 188, 191

CreateDTDEntity 185
CreateDTDNotation 185
CreateElement 131
CreateElementNS 132
CreateEntityReference 132
CreateProcessingInstruction 132
CreateTextNode 133
CSS 273

D

Data 151, 214
DefaultType 202
DefaultValue 203
Delete 140, 146
DeleteData 152
display-align 416
DisplayName 264
DocType 133
Document 86
DocumentElement 87, 133
DomImplementation 133

E

ecXxx 102
ElementText 158
Empty 141, 146
empty-cells 417
end-indent 417
ends-row 418
Enumeration 203
ErrorCount 53, 87
Errors 54, 87, 251
Exchange 141
external-destination 418
ExternalDTD 186
external-graphic 370

F

Filter 251
FindElement 159
FirstChild 111
flow 371
flow-name 419
font-family 419
font-size 420
font-size-adjust 420
font-stretch 421
font-style 422
font-variant 422
font-weight 423
ForceOwnerDocument 112
FormattedOutput 88, 134, 273, 283
FullEndTag 159

G

GetAttribute 160
GetAttributeNode 160
GetAttributeNodeNS 161
GetAttributeNS 161
GetChildElementsByTagName 162
GetChildNodesByNodeType 112
GetElementsByTagName 135, 162
GetElementsByTagNameNS 135, 163
GetElementsByTagNameWithAttribute 163
GetErrorMsg 54, 89
GetNamedItem 147
GetNodesByNodeType 114
GetVariable 252

H

HasAttribute 164
HasAttributes 115
HasChildNodes 115
HasFeature 106
height 423

HPixelsPerInch 264
HSlideFactor 277
HSlidePosition 277
HTMLDocument 274
hyphenate 424
hyphenation-character 425
hyphenation-keep 425
hyphenation-push-character-count 425
hyphenation-remain-character-count 426

I

id 426
IdAttribute 90, 136
IgnoreCase 90, 136, 252
IgnoreEndTag 164
ImportNode 136
InCharSet 55
IndexOf 141, 147
inline 371
Insert 142, 147
InsertBefore 115
InsertData 152
IsAfter 116
IsExternalDTD 186
IsPE 206
Item 142, 148

K

keep-together 427
keep-with-next 428
keep-with-previous 428

L

language 429
LastChild 116
layout-master-set 372
leader 372
leader-length 429
leader-pattern 430
Length 142, 148, 152
letter-spacing 431
LevelCode 117
line-height 431
line-height-shift-adjustment 432
list-block 374
list-item 375
list-item-body 376
list-item-label 376
LoadDataSource 90
LoadMemory 91
LocalName 117

M

margin-bottom 433
margin-left 434
margin-right 434
margin-top 435
master-name 435
Move 143

N

Name 155, 186, 188, 194, 199, 203, 206, 209
NamespaceURI 117
NextSibling 118
NoCharReplacement 136
NodeID 118
NodeName 118
NodeStringType 120
NodeType 121
NodeValue 122
Normalize 164
NormalizeData 55, 92
NotationName 206
number-columns-repeated 436

O

Occurs 191, 194
odd-or-even 436
OnApplyStyleEnd 253
OnApplyStyleStart 253
OnAttribute 57, 253
OnCDATASection 58, 254
OnCharData 58
OnComment 59, 254
OnDocTypeDecl 59
OnDTDAttDefinition 60
OnDTDAttlistEnd 61
OnDTDAttlistStart 62
OnDTDConditionalEnd 62
OnDTDConditionalStart 62
OnDTDElementContent 63
OnDTDElementEnd 63
OnDTDElementGroupEnd 64
OnDTDElementGroupStart 65
OnDTDElementStart 65
OnDTDEnd 66
OnDTDEntityExternal 66
OnDTDEntityInternal 66
OnDTDExternalEnd 67
OnDTDExternalStart 67
OnDTDNotation 67
OnDTDStart 68
OnElementEnd 254
OnElementStart 255
OnEndDocument 68
OnEndElement 69
OnFoProperty 255
OnFormatObjectEnd 256
OnFormatObjectStart 256
OnFunction 93, 137
OnFunctionAvailable 94, 137
OnInvalidDocument 94
OnNonXMLEntity 69
OnPreserveSpace 70, 96
OnProcessingInstruction 70, 257
OnQueryFilter 257
OnResolveElement 258
OnResolveElementCode 258
OnResolveEntity 71, 96
OnStartDocument 71
OnStartElement 72
OnSystemProperty 97, 137
OnText 259
OnVariableLookup 97, 138
orphans 437
OutputEncoding 274, 284
OutputMode 275
overflow 437
OwnerDocument 122
OwnerElement 155

P

padding-after 438
padding-before 439
padding-bottom 440
padding-end 441
padding-left 442
padding-right 443
padding-start 444
padding-top 445
PageCount 277
PageCurrent 277
page-height 445
page-number 377
page-position 446
page-sequence 378
page-sequence-master 379
page-width 447
ParentNode 123
ParseDataSource 72
ParseMemory 73
Password 73, 97
Prefix 123
PreviousSibling 123
Print 278
provisional-distance-between-starts 448

provisional-label-separation 448
PublicID 206, 209

R

RaiseErrors 73, 98, 259
region-after 380
region-before 381
region-body 382
region-end 383
region-name 449
region-start 384
REL_XX 190
Relation 195
Release 124
RemoveAll 124
RemoveAttribute 165
RemoveAttributeNode 165
RemoveAttributeNS 166
RemoveChild 124
RemoveNamedItem 148
repeatable-page-master-alternatives 385
repeatable-page-master-reference 385
Replace 143, 148
ReplaceChild 125
ReplaceData 153
Reset 264
root 385
RTFDocument 281
rule-thickness 450

S

SaveToFile 98, 265
scaling 450
scaling-method 451
SelectBoolean 166
SelectInteger 167
SelectNodes 168
SelectNumber 170
SelectSingleNode 170
SelectString 171
SetAttribute 171
SetAttributeNode 172
SetAttributeNodeNS 173
SetAttributeNS 173
SetNamedItem 149
SetVariable 259
show-destination 451
simple-page-master 386
single-page-master-reference 386
Sort 144
space-after 452
space-before 452
space-end 453
space-start 453
Specified 155
SplitText 177
src 454
start-indent 454
starts-row 454
static-content 387
StyleData 260
StyleURL 260
SubstringData 153
Supports 125
SystemID 207, 209

T

table 387
table-body 388
table-cell 389
table-column 390
table-row 390
TagName 174
Target 214
Text 126
text-align 455
text-align-last 456
text-indent 457
title 391

TrackAttributes 98
TrackedAttributes 99
TrackedElements 99
TrackElements 99
TXpAttribute 154
TXpCDATASection 182
TXpCharacterData 150
TXpComment 178
TXpDocument 127
TXpDocumentFragment 107
TXpDocumentType 183
TXpDOMException 101
TXpDOMImplementation 104
TXpDTDAttDefinition 200
TXpDTDAttList 196
TXpDTDConditional 210
TXpDTDElement 187
TXpDTDElementContent 192
TXpDTDElementGroup 189
TXpDTDEntity 204
TXpDTDNotation 208
TXpElement 156
TXpEntityReference 212
TXpFilterBase 263
TXpFilterHTML 272
TXpFilterPrint 276
TXpFilterRTF 280
TXpFilterXML 282
TXpNamedNodeMap 145
TXpNode 139
TXpNodeList 139
TXpObjModel 85
TXpParser 52
TXpProcessingInstruction 213
TXpText 175
TXpXSLProcessor 244
TXpXxx 119, 120, 122

U

UserName 73, 99

V

V_XXX 95
ValidateDocument 100
Value 155, 207
VPixelsPerInch 265
VSlideFactor 279
VSlidePosition 279

W

widows 457
width 458
word-spacing 458
wrap-option 459

X

XMLDocument 144, 284
XmlDocument 100, 126
XmlDTD 138
XMLValidHTML 275
XpObjModel 261
xsl
    apply-imports 295
    apply-templates 296
    attribute 298
    attribute-set 300
    call-template 301
    choose 303
    comment 304
    copy 304
    copy-of 305
    decimal-format 306
    element 308
    fallback 310
    for-each 311
    if 313
    import 316
    include 318
    key 318

message 320
namespace-alias 321
number 323
otherwise 328
output 329
sort 339
strip-space 342
stylesheet 343
template 345
text 349
transform 350
value-of 350
variable 351
when 352
with-param 353
XslProcessorApplyStyleEnd 266
XslProcessorApplyStyleStart 266
XslProcessorAttribute 266
XslProcessorCDATASection 267
XslProcessorComment 267
XslProcessorElementEnd 268
XslProcessorElementStart 268
XslProcessorFoProperty 268
XslProcessorFormatObjectEnd 269
XslProcessorFormatObjectStart 269
XslProcessorProcessingInstruction 269
XslProcessorQueryFilter 270
XslProcessorResolveElement 270
XslProcessorResolveElementCode 271
XslProcessorText 271

Subject Index

A

accessing
  character data 151
  Document Element 133
  Document Type Definition (DTD) 133
adding
  element attribute 171, 173
  footer to page 380
  heading to page 381
  left sidebar to page 384
  main content section to page 382
  right sidebar to page 383
aliasing namespace 321
aligning inline content 455
alignment, specifying for inline areas 392
allowing hyphenation 424
alternative processing path 310
appending, data to node 151
appendix 461
applying
  stylesheet to document 239, 250, 266
  template rules to set of nodes 296
area 357, 481
  forcing to be first 412
  forcing to be last 411
  setting bottom margin width 433
  setting left margin width 434
  setting right margin width 434
  setting top margin width 435
  specifying height 423
  specifying line height 431
area model 357
area tree 481
aspect ratio, preserving 450
attribute
  accessing 110, 160, 161
  accessing list of 99
  adding existing 173
  adding to element 171
  creating in current element 298
  creating new 171, 173
  creating node 129
  creating node within namespace 129
  declaration name 203
  defined 481
  defining named set 300
  definition creating 197
  detecting 57, 115, 164
  detecting when sent to result tree 253, 266
  enumeration values 203
  having specific name 160, 161
  identifying owner element 155
  identifying type 201
  name 155
  owner 155
  removing named 165, 166
  removing node 165
  retrieving from namespace 161
  retrieving node 160
  retrieving value 160, 161
  returning name 155
  returning value 155
  selecting 168
  setting default 202
  setting default value 203
  specified in element declaration 155
  tracking 98

attribute list
encountering declaration end 61
encountering declaration start 62
encountering values 60
attribute value template 481

B

background color, specifying 394
background image
specifying initial horizontal position 395
specifying initial vertical position 396
specifying whether fixed or scrolling 393
backround, specifying image drawn on 394
blank page condition, defining 398
block area
defined 481
described 357
block-level area
specifying space after 452
specifying space before 452
body region, retrieving 283
border
specifying before edge width 400
specifying bottom edge width 402
specifying color for after edge 398
specifying color for before edge 400
specifying color for bottom edge 401
specifying color for end edge 403
specifying color for left edge 405
specifying color for right edge 406
specifying color for start edge 408
specifying color for top edge 410
specifying width of after edge 399
specifying width of end edge 404
specifying width of left edge 405
specifying width of right edge 407
specifying width of start edge 409
specifying width of top edge 410
buffer size
defining 53
DOM 86

C

calling
attribute found 57
CDATA found 58
character data found 58
comment found 59
conditional end 62
conditional start 62
document start 71
DTD attribute definition found 60
DTD attribute list end 61
DTD attribute list start 62
DTD conditional end 62
DTD conditional start 62
DTD element group end 64
DTD element group start 65
DTD element start 65
DTD end 66
DTD external end 67
DTD external entity 66
DTD external start 67
DTD internal entity 66
DTD notation found 67
DTD start 68
element content 63
element end 69
element start 72
processing instruction found 70
cascading stylesheet (CSS), specifying whether data is created within HTML document 273
case-sensitivity
defined 29
ignoring while searching 90, 136
specifying 252

CDATA
controlling output format 129
creating 130, 158
encountering 58
passing to result tree 254
CDATA section 482
cell, specifying whether it starts a row 454
changing, XSL command name 258
character data 481
accessing 151
appending 151
deleting 152
determining length 152
encountering 58
extracting 153
inserting 152
removing 152
replacing 153
returning length 152
character set
defined 29
determining 55
character, controlling replacement 136
child area, specifying alignment 416
child element
appending 110
detecting 115
finding 159
removing 124
removing all 124
replacing 125
child node
accessing 111
appending 110
determining number 115
inserting before 115
removing all 124
removing specified 124
replacing 125
retrieving by node type 114
retrieving list of matching 112
returning first 111
child node (continued)
returning last 116
returning text 126
clearing
filter's result tree 264
variables from stylesheet 250
cloning node 111, 136
code, exception 102
collision, handling 228
color
specifying for area 413
specifying for background 394
column
defining 390
specifying numerical identifier 414
specifying width 414
comment 481
creating 130
encountering 59
sending to result tree 254, 267
comment node, creating in result tree 304
condition, specifying for a repeating page sequence 369
conditional logic, using 230
conditional section
creating 184
encountering end 62
encountering start 62
specifying type 211
conditionally repeated page, specifying 385
content 481
adding main section to page 382
defining for page 378
formatting 362
formatting table cell 389
formatting within a table 387
grouping related 368
specifying height 415
specifying width 415
controlling hyphenation 425
creating
attribute 171

creating (continued)
attribute in result tree 298
CDATA node 158
comment node in result tree 304
document object 105
DTD 105
DTD attribute definition 197
DTD attribute list 184
DTD conditional section 184
DTD element content 190
DTD element declaration 185
DTD element group 188, 191
DTD entity declaration 185
DTD notation declaration 185
element 158
leader 372
processing instruction node 338
rule 372
text node 158, 349, 350
current node 482
current node list 482
current template rule 482

D

data
appending 151
deleting 152
inserting 152
parsing 73, 91
replacing 153
data source
loading 72, 90
parsing 72, 90
retrieving 72, 90
declaration
attribute list 62
defined 482
declaring
global parameter 336
global variable 351
local variable 351
stylesheet 343, 350
defining
an enumeration, identifier, or decoration for list item 376
blank page condition 398
buffer size 53, 86
characters and symbols used by format number function 306
column within a table 390
content for a page 378
content for a sequence of pages 378
content for list item 376
entity name 206
flow name 419
group of items in a list 374
link 367
name 209
named set of attributes 300
notation name 206
odd or even condition 436
page layout 386
page position condition 446
parameter entity 206
public ID 206
repeated sequence of pages 385
rows contained in table 388
set of flow objects 371
set of page-layout definitions 372
set of page-sequence definitions 372
single-page page sequence 386
static content on one or more pages 387
template rule 345
title for document 391
top node of formatting object tree 385
deleting
character data 152
node 140, 146
designing
layout 358
sequence of pages 359
destroying current object model 86

detecting attributes of element 164
determining
  current page in the print filter 277
  HTML character encoding 274, 284
determining, character set 55
displaying destination resource 451
document
  clearing 86
  creating 79, 105
  creating fragment 130
  defining title 391
  detecting end 68
  detecting invalid 94
  detecting start 71
  force owner 112
  loading 72, 76
  navigating 78
  normalizing 55, 92
  owner 122
  providing direct access 87
  returning formatted 138
  returning name 186
  returning number of printed pages 277
  saving to disk 265
  saving to file 76, 98
  supporting large 4
  validating against stored DTD 100
document element 482
Document Object Model (DOM)
  defined 42, 75, 482
  validation error codes 462
document type
  creating 131
  encountering 59
Document Type Definition (DTD)
  accessing external 186
  creating 105, 130
  creating attribute list 184
  creating element declaration 185
  creating entity declaration 185
  creating notation declaration 185
  defined 31
Document Type Definition (DTD) (continued)
  detecting end 66
  detecting start 68
  identifying external 186
  name 186
  providing access to 133, 186
  returning formatted 138
DOM 483
DOM implementation
  accessing 133
  supporting specific features 106
DTD 483

E

element 483
  accessing attributes 110
  accessing root 133
  accessing text 126
  adding attribute 171, 172
  cloning 111
  creating 131
  creating DTD attribute definition 197
  creating DTD group 191
  creating within namespace 132
  declaring content 192
  defined 25
  detecting attributes 115, 164
  detecting child elements 115
  detecting end 63
  detecting start 65
  finding 159, 162, 163
  forcing output of full end tag 159
  forcing owner 112
  ignoring end tag 164
  instantiating within result tree 308
  local name 117
  name 118, 174, 199
  normalizing 164
  owner document 122

element (continued)
removing attribute 165
removing named attribute 165
retrieving attribute 160
retrieving by name with attribute 163
retrieving child by name 162
retrieving descendants by name 162
retrieving within namespace 163
returning associated text 158
returning by tag name 135
returning within namespace 135
selecting 166, 167, 170
sending to result tree 254, 255, 268
specifying relation 195
tracking 99
element content
creating 190
defined 26
detecting 63
element declaration
creating an element group 188
identifying content type 188
name 188
providing access to name 188
element group
creating 191
detecting end 64
detecting start 65
member name 194
specifying how it occurs 191, 194
encountering
attribute list 60
CDATA section 58
character data 58
comment 59
document type 59
fuction 137
processing instruction node 257
entity 483
defined 35
defining name 206
entity (continued)
defining notation name 206
detecting external 66
detecting internal 66
general 35, 204
identifying value 207
public ID 206
resolving 71
system ID 207
entity reference
creating 132
defined 483
error
count 53, 87
number of 53, 87
raising 73, 98
retrieving 251
retrieving message 54, 87, 89
error codes
exception 464
validation 462
error message, returning 54, 87, 89
event-driven parsing 43
character sets 44
establishing event handlers 46
handling errors 50
initiating parse 49
memory usage 51
example
TextView 12
XMLEdit 12
XMLTest 12
EXAMPLESCBUILDRX*.* 12
EXAMPLESDELPHI*.* 12
EXAMPLESXMLDOCS*.* 12
exception error codes 102, 464
expression
referencing variable 97, 138
XPath 286
Extensible Markup Language (XML) 23, 483
Extensible Style Language (XSL) 484

external entity
  detecting 66
  resolving location 96
  user name 99
external resource, specifying for a link 418

F

fallback, specifying 310
feature, identifying supported 106, 125
files supplied 9
filter
  clearing result tree 264
  defined 262
  explained 239
  identifying component 251
  querying name 257
  reference 239
  specifying name 264
  using 241
finding
  attribute 160, 161
  element 159, 162, 163
flow
  defined 484
  defining name 419
  defining page content 371
font
  selecting style from family 422
  selecting width from family 421
  specifying ideal ratio between x-height and size 420
  specifying list of family name 419
  specifying size when set solid 420
  specifying small-caps version 422
  specifying weight 423
footer, adding to page 380
forcing
  area to be first 412
  area to be last 411
format
  controlling CDATA 129
  returning document 138
format number function, defining characters and symbols used 306
formatting 484
  content 362
  content placed in table cell 389
  content within a table 387
  explained 215
  graphics 366
  inline content 371
  leaders and rules 365
  links 366
  lists 362
  output 88, 134
  table 364
  text 362
formatting object
  defined 484
  explained 355
  processing 269
  processing command 256
formatting property
  defined 484
  explained 356
  predefined colors 468
FTP
  loading document 90
  saving document 98
  specifying password 97
function, encountering 94, 137

G

general entity 35, 204
global variable 236, 351
graphic
  formatting 366
  retrieving external 370

graphic (continued)
  scaling 451
  sizing 451
grouping
  related content 368
  table cells into rows 390

H

handling template collisions 228
header, adding to page 381
HELP*.HLP 9
horizontal position, controlling 277
HTML 23
HTML document
  determining character encoding 274
  indicating line breaks 273
  output 333
  retrieving 274
  specifying content 275
  specifying inclusion of end tags 275
HyperText Markup Language (HTML) 23, 484
hyphenation
  allowing 424
  controlling 425
  specifying character 425
  specifying number of characters after 425
  specifying number of characters before 426

I

ID attribute name 90, 136
IDE integration 15
identifier
  formatting object 426
  node location 117
identifying
  attribute owner 155
  attribute type 201
  content type 188
  entity value 207
  filter component 251
  import precedence 228, 316
  import tree 228, 316
  master 435
  owner document 122
  public ID 206, 209
  region 449
  supported features 125
  system ID 207, 209
  value 207
ignoring end tags 164
import precedence 484
import tree 484
importing
  node 136
  stylesheet 227, 316
imposing
  keep-together conditions 427
  keep-with-next conditions 428
  keep-with-previous conditions 428
including stylesheet 318
indent, specifying 417, 454, 457
inline area 357, 484
  specifying space after 453
  specifying space before 453
inline content, formatting 371, 455
inserting
  data 152
  node 142
installing 15
instantiating
  template for each node 311
  template if boolean expression evaluates to True 313

invalid document 94
invoking named template 234, 301

K

keep-together condition, imposing 427
keep-with-next condition, imposing 428
keep-with-previous condition, imposing 428
key 318

L

language, specifying which to use 429
layout, defining 386
layout, designing 358
leader
  creating 372
  formatting 365
  specifying fill 430
  specifying length 429
  specifying pattern 430
left sidebar, adding optional to page 384
link
  defining 367
  formatting 366
  specifying external resource 418
list
  defining group of items 374
  formatting 362
list item 376
  containing list label and body 375
  defining an enumeration for 376
  defining an identifier for 376
  defining content 376
list label, forming list item 375
literal result element 484
literal text, outputting to result tree 349
local variable, specifying 351
location, external entity 96

M

maintainer 485
mapping namespace 321
margin
  specifying bottom 433
  specifying left 434
  specifying right 434
  specifying top 435
markup 23, 485
master
  identifying 435
  selecting 435
metalanguage 40, 485
mixed content 26
moving node 143

N

name
  document 186
  ID attribute 90, 136
  namespace 123
  node 118
  prefix 123
  user 99
named key, declaring 318
named template, declaring 345
namespace 485
  aliasing 321
  for current node 117
  mapping to different 321
namespace prefix 485
namespace URI 485
naming conventions 19
node
  accepting as owning document 112
  accessing attributes 110
  accessing character data 151

node(continued)
accessing child 111
adding contents to current list 140
adding to end of children list 110
adding to map 146, 149
adding to node list 140
appending data 151
cloning 111, 136
copying to node list 140
creating attribute 129
creating CDATA 130
creating comment 130
creating document type 131
creating element 131
creating element within specified namespace 132
creating entity reference 132
creating fragment 130
creating processing instruction 132
creating text node 133
decrementing reference count 124
deleting 140, 146
determining number of child nodes 115
determining position 116
determining presence of attributes 115
exchanging 141
extracting range of characters 153
identifying owner document 122
importing 136
increasing reference count 110
indicating type 121
inserting 142
inserting before 115
inserting data 152
inserting into map 147
instantiating template 311
location 117
moving 143
name 118
performing a deep copy 305
performing a shallow copy 304
removing all 124, 141
node(continued)
removing all from map 146
removing attribute 165
removing from map 148
removing specified 124
replacing 125, 143, 148
retrieving by name 147
retrieving by node type 114
retrieving from map 147
retrieving list of 112
returning as well-formed 126
returning first 111
returning index in node map 147
returning index of 141
returning last child 116
returning length 142
returning length of character data 152
returning list as XML fragment 144
returning local name 117
returning namespace for current 117
returning next sibling 118
returning number in list 142
returning number in map 148
returning parent 123
returning position 142, 148
returning prefix 123
returning previous sibling 123
returning text 126
returning value 122
selecting 168
sorting 144, 339
switching position 141
type 120
user-definable ID 118
node map
adding 146
adding node 149
inserting node 147
removing all nodes 146
removing nodes from 148

node map(continued)
returning node index 147
returning number of nodes 148
normalizing data 55, 92
notation
name 209
public ID 206, 209
system ID 207, 209

O

object model
defined 42
destroying current 86
node location 117
returning document object 86
writing to FTP site 98
writing to local file 98
on-line help 19
orphaned lines, specifying 437
output
formatting 88, 134
improving readability 88, 134
indenting 88, 134
line breaks 88, 134
overflow, specifying 437
overriding global variable value 259
owner document 122
owner element 155

P

padding
specifying width on after-edge 438
specifying width on before-edge 439
specifying width on bottom-edge 440
specifying width on end-edge 441
specifying width on left-edge 442
specifying width on right-edge 443
specifying width on start-edge 444
specifying width on top-edge 445
page
defining layout 386
determining current 277
layout 358
printing 278
retrieving number 377
sequence 358
specifying height 445
specifying position 446
specifying sequence 379
specifying width 447
page-layout, defining set 372
page-master selection
specifying blank 398
specifying odd or even 436
specifying page position 446
page-sequence
defining repeated 385
defining set 372
page-viewport-area 485
parameter
declaring 336
passing to named template 301, 353
parameter entity
defined 35
defining 206
parsed data 29
parser 53
parsing
in-memory document 91
reporting errors 87
passing
CDATA to result tree 254
parameter to named template 353
processing instruction to result tree 269
password, for FTP site 97
pixel
returning number that will scroll horizontally 277
returning number that will scroll vertically 279

pixel (continued)
specifying number per inch horizontally 264
specifying number per inch on output document 265
preserving intrinsic aspect ratio 450
preserving whitespace-only text nodes 337
printing selected page 278
processing
formatting object 269
formatting object command 256
XSLT 217
processing instruction
content 214
creating node 132, 338
defined 213
encountering 257
passing to result tree 269
target 214
prolog 485

Q

querying
filter name 257
XSL processor for filter name 270

R

README.HLP 9
region, identifying 449
region-viewport-area 486
removing
all nodes 124, 141
named attributes 165, 166
node attributes 165
specified node 124
white space 55, 92
repeated sequence of pages, defining 385
replacing
ASCII characters 136
character data 153
child node 125
node 143, 148
resolving
stylesheet element code 258
XSL processor element 270
XSL processor element code 271
result tree 486
clearing 264
defined 215, 240
writing to output document 329
retrieving
attribute value 160
body region of XML document 283
elements by name 162
external graphic 370
HTML document 274
node attribute 161
node from map 147
page number of current inline-area 377
rendered RTF document 281
stylesheet errors 251
value of global variable 252
XML document 284
returning
attribute value 155
boolean result 166
document object 86
document text 126
element text 158
expression as integer 167
first matching element 170
index of node 141
length of character data 152
name of attribute 155
node length 142
node list 142
node position 142

returning (continued)
node text 126
number of pages in document 277
number of pixels that will scroll horizontally 277
number of pixels that will scroll vertically 279
parent node 123
position of node 148
prefix of node name 123
previous sibling of node 123
result as a double 170
result as a string 171
root 133
specified error message 89
tag name 174
well-formed document 126
right sidebar, adding to page 383
root
defined 357
defining 385
root element
defined 25, 486
providing direct access 87
row
creating 390
defining 388
RTF document, retrieving rendered 281
rule
creating 372
formatting 365
specifying thickness 450

S

saving document to disk 265
scaling, preserving intrinsic aspect ratio 450
selecting
font style from family 422
font width from family 421
selecting (continued)
master 435
node 168
sending
attribute to result tree 253, 266
comment to result tree 254, 267
element to result tree 254, 255, 268
text element to result tree 259
text to result tree 271
setting
bottom margin width of a box 433
left margin width of a box 434
right margin width of a box 434
top margin width of a box 435
SGML 23, 486
sibling, returning next 118
sidebar
left 384
right 383
simple content 26
single-page page sequence, defining 386
sorting nodes 144, 339
specifying
alignment point for inline areas 392
an in memory stylesheet 260
area's background color 394
border bottom edge width 402
border color for after edge 398
border color for before edge 400
border color for bottom edge 401
border color for end edge 403
border color for left edge 405
border color for right edge 406
border color for start edge 408
border color for top edge 410
border width of after edge 399
border's before edge width 400
border's end edge width 404
box line height 431
case-sensitivity 136, 252
child area alignment 416

specifying (continued)
color for area 413
column width 414
condition for repeating page sequence 369
conditionally repeated page 385
content height 415
content width 415
element relation 195
external resource for a link 418
filter name 264
font size when set solid 420
font weight 423
height of boxes 423
HTML content 275
ideal ratio between x-height and size of font 420
ident 417
image drawn on area's background 394
inclusion of end tags in HTML document 275
indent 454
initial horizontal position of background image 395
initial vertical position of background image 396
language to use with local specific services 429
leader fill 430
leader length 429
leader pattern 430
list of font family names 419
node sort order 339
number of characters after hyphenation 425
number of characters before hyphenation 426
number of pixels per inch horizontally 264
number of pixels per inch on output document 265
numercial identifier for column 414
orphaned lines 437
overflow 437
specifying (continued)
padding width 438
padding width on before-edge 439
padding width on bottom-edge 440
padding width on left-edge 442
padding width on start-edge 444
padding width on top-edge 445
padding width onn end-edge 441
padding width right-edge 443
page height 445
page position 446
page sequence 379
page width 447
password 97
rule thickness 450
space after block-level area 452
space after inline area 453
space before block-level area 452
space before inline area 453
spacing behavior between text characters 431
stylesheet to apply to document 260
template to be instantiated 328
text indentation 457
type of conditional section 211
Unicode character used 425
unique identifier 426
URI reference to an external resource 454
whether a cell starts a row 454
whether a table cell end a row 418
widowed lines 457
width of border's left edge 405
width of border's right edge 407
width of border's start edge 409
width of border's top edge 410
Standard Generalized Markup Language (SGML) 23, 486
static content, defining 387
stripping whitespace-only text node 342
stylesheet 486
applying to document 250, 266
clearing variables 250

stylesheet (continued)
creating 217
declaring 343, 350
identifying filter component 251
importing 227, 316
including in current 318
overriding 295, 347
programming 230
resolving element code 258
retrieving errors 251
specifying 260
suggested reading 21
system requirements 8
system-property function
encountering 97
encountering XSLT system property 137

T

table
defining column 390
defining rows 388
formatting 364
formatting content 387
table cell
controlling borders with no visible content 417
formatting content 389
grouping into rows 390
specifying whether ends a row 418
tag 24, 486
returning name 174
technical support 20
template
applying to nodes 296
calling 234, 301
defining 345
handling collisions 228, 348
instantiating for each node 311
instantiating if boolean expression evaluates to True 313
template (continued)
invoking overriden rule 295
naming 345
using multiple 221
text
formatting 362
specifying indentation 457
specifying spacing behavior 431
text element, sending to result tree 259
text node
condensing adjacent 164
creating 133, 158, 350
splitting into two 177
title, defining for document 391
top level element 486
tracking
attributes in a document 98
elements 99
transformation 486

U

Unicode 486
Universal Resource Identifier (URI) 487
Universal Resource Locator (URL) 487
Universal Resource Name (URN) 487
unparsed data 29
URI reference, specifying to an external resource 454
user function, referencing 93
user functions 137
user name 99
user-definable ID, specifying 118
using conditional logic 230

V

validating a document 100
validating processor 487

value
attribute 155
node 122
variable
clearing from stylesheet 250
declaring global 236, 351
declaring local 351
overriding value 259
referencing in an expression 97
retrieving value 252
using 236
variable, referencing in an expression 138
vertical position, controlling 279
viewport 487

W

W3C 24, 487
well-formed 487
defined 28
returning document 126
white space
defined 29
preserving 81, 96, 337
removing extra 55, 92, 342
whitespace-only text node
preserving 337
stripping 342
widowed lines, specifying 457
World Wide Web Consortium 24, 487
writing result tree to output document 329

X

XML 487
communicating between business objects 40
creating new languages 40
defined 23
Document Object Model (DOM) 42
driving web content 40
XML (continued)
electronic exchange of information 39
event-driven parsing 41
overview 23
saving configuration data 38
sharing data 40
syntax 25
temporary data storage 39
XML document, validating 100
XML fragment, returning 144
XML Path Language (XPath)
defined 487
detecting user function 93
using 83, 166, 167, 170
XML processor 488
XpAbout.pas 9
XpAboutw.pas 9
XPath
expression 286
functions 466
patterns 293
XPath expression
returning as a string 171
returning as integer 167
returning boolean result 166
returning first matching element 170
returning result as a double 170
XpBase.pas 9
XpChrFlt.pas 9
XpDOM.pas 9
XpExcept.inc 9
XpExcept.pas 10
XpFltBas.pas 10
XpFltHTM.pas 10
XpFltPrt.pas 10
XpFltRTF.pas 10
XpFltXML.pas 10
XpInet.pas 10
XpParser.pas 10
XpReg.pas 10
XpXSLPrc.pas 10

XSL
  changing command name 258
  defined 488
  overview 215
  processor 239
  using components 241
XSL area model 357
XSL processing
  beginning 253
  completing 253
XSL processor
  applying style 250
  property tag codes 477
XSL processor (continued)
  querying for filter name 270
  raising errors 259
  raising ONXSL Message 320
  resolving element 270
  resolving element code 271
  sending text to result tree 271
  setting stylesheet 260
  tag codes 474
  using 241
  without a filter 244
XSL Transformation Language (XSLT) 488
XSLT reference 285